SAMS Teach Yourself

- 全球销量逾百万册的系列图书
- 连续十余年打造的经典品牌
- 直观、循序渐进的学习教程
- 掌握关键知识的最佳起点
- “Read Less, Do More”（精读多练）的教学理念
- 以示例引导读者完成最常见的任务

每章内容针对初学者精心设计，1小时轻松阅读学习，
24小时彻底掌握关键知识

每章案例与练习题助你轻松完成常见任务，
通过实践提高应用技能，巩固所学知识

Unity游戏开发入门经典

[美] Mike Geig 著
古宏霞 译

人民邮电出版社
POSTS & TELECOM PRESS

目　录

Unity游戏开发入门经典

[美]Mike Geig　著
古宏霞　译

人民邮电出版社

北京

图书在版编目（**CIP**）数据

Unity游戏开发入门经典/（美）吉格（Geig，M.)著；古宏霞译.--北京：人民邮电出版社，2015.2

ISBN　978-7-115-37912-2

Ⅰ.①U…　Ⅱ.①吉…②古…　Ⅲ.①游戏程序—程序设计　Ⅳ.①TP311.5

中国版本图书馆CIP数据核字（2015）第005391号

◆著　[美] Mike Geig

译　古宏霞

责任编辑　傅道坤

责任印制　张佳莹

◆人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

邮编　100164　　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

北京艺辉印刷有限公司印刷

◆开本：787×1092　1/16

印张：20

字数：491千字　　2015年2月第1版

印数：1-3000册　　2015年2月北京第1次印刷

著作权合同登记号　图字：01-2013-9203号

定价：49.00元

读者服务热线：**(010)81055410**　印装质量热线：**(010)81055316**

反盗版热线：**(010)81055315**

内容提要

Unity 游戏引擎是由 Unity Technologies 公司开发的一个让玩家轻松创建诸如三维视频游戏、建筑可视化、实时三维动画等内容的跨平台综合游戏开发工具。当前很多热门的游戏（比如Temple Run）都是使用Unity开发的。

本书采用直观易懂的方法，为零基础的读者讲解了游戏开发的基本知识，并通过4个完整的游戏示例来演示Unity游戏开发的方法和技巧。本书分为24章，其内容包括Unity简介、创建和使用游戏对象、高效使用Unity图形资源管线、在3D对象上应用着色器和纹理、利用Unity的地形和环境工具集来生成逼真的游戏世界、使用预制件（prefabs）快速创建可重用的游戏对象、创建直观的游戏用户界面、使用Unity的Shuriken 离子系统创建游戏特效、充分使用Unity全新的Mecanim动画系统、在游戏中集成2D/3D环境音效、使用移动设备加速计和多触摸显示屏、将桌面游戏移植道移动平台上，以及部署游戏。

本书适合对使用Unity进行游戏开发感兴趣的零基础读者学习；有过其他游戏平台开发经验，打算向Unity平台转移的读者也可以通过本书迅速上手。

关于作者

Mike Geig 既是一位经验丰富的教师，也是一位经验丰富的游戏开发人员，他在这两个领域都具有很深的造诣。Mike 目前在美国斯塔克州立学院和克利夫兰艺术学院教授游戏设计和开发，他还担任Unity Technologies 的屏幕录制师，并且是Unity Learn（学习）部门的成员。他的Pearson 视频*Game Development Essentials with Unity 4 LiveLessons*是学习Unity的非常重要的作品。Mike激情洋溢，在Facebook上具有超过100万的“同党”。

献辞

献给我的父亲：您的一切，都值得我学习。

致谢

非常感谢帮助我编写这本书的每一个人。

首先并且最重要的是，感谢Kara使我一直走在正轨上。我不知道在这本书问世时我们将会谈论些什么，但无论谈论什么，你可能都是对的。爱你，宝贝。

Link和Luke：我们应该让妈妈轻松一小会儿。我认为她快崩溃了。

感谢我的父母。当我自己现在也是一位父亲时，我认识到你们把我抚养长大有多艰难。谢谢您们。

感谢Angelina Jolie。由于您在电影Hackers（1995年）中扮演的角色，我决定学习如何使用计算机。您低估了自己当时对一个10岁孩子的影响，您是一位精英！

感谢牛肉干的发明者：历史可能忘记了你的名字，但是一定不会忘记你的产品。我非常喜欢这种食品，谢谢你！

感谢我的技术编辑：Valerie、Jim和Tim。你们的编校工作和深入的见解在使本书变得更好的过程中起着至关重要的作用。

谢谢你，Laura，是你说服我编写本书。还要谢谢你在GDC给我买午餐，我感觉那份午餐是一日三餐中最好的，它对我写完这本书起到了特别的作用。

最后，还要感谢Unity Technologies。如果你从未创建过Unity游戏引擎，本书将显得非常突兀，并且会让你困惑。

前言

Unity 游戏引擎极其强大，因此受到专业人员和业余的游戏开发人员青睐。编写本书的目的是为了使读者能够尽快地熟悉 Unity 并使用它来工作。同时本书还介绍了游戏开发的一些基本原理。与其他只介绍特定主题或者利用全部篇幅讲解单独一款游戏的图书不同，本书在介绍大量主题的同时仍然设法包含了4款游戏。在你读完本书后，所掌握的不仅仅是Unity游戏引擎方面的理论知识，还将拥有与之配套的游戏开发技能。

本书读者对象

本书适合于任何希望学习如何使用 Unity 游戏引擎的人。无论你是一名学生还是开发人员，都能从本书中学到自己想要的知识。本书不要求读者具备任何游戏开发知识或经验，因此，如果你是第一次涉足创建游戏的艺术这个领域，也不要担心。花一些时间好好享受一下，你将立即开始投入到学习中去。

本书组织结构与主要内容

本书分为24章，每一章应该花大约1小时的时间完成。本书各章分别介绍以下内容。

第**1**章，**“Unity** 简介**”**：本章将让你开始运行Unity 游戏引擎的多种组件。

第**2**章，**“**游戏对象**”**：本章将教会你如何使用 Unity 游戏引擎的基本构件，即游戏对象。你还将学习坐标系统和变换。

第**3**章，**“**模型、材质和纹理**”**：在本章中，你将学习在对材质应用

着色器和纹理时，使用Unity的图形资源管线，还将学习如何对各种3D对象应用这些材质。

第**4**章，**“**地形**”**：在本章中，你将学习使用 Unity 的地形系统雕刻游戏世界。在你四处搜寻并创建独特、绝妙的风景时，不要害怕弄脏你的手。

第**5**章，**“**环境**”**：在本章中，你将学习对你雕刻的地形应用环境效果，是时候栽一些树了！

第**6**章，**“**灯光和摄像机**”**：本章将非常详细地介绍灯光和摄像机。

第**7**章，**“**第**1**款游戏：**Amazing Racer”**：是时候开发你的第一款游戏了。在本章中，你将创建 Amazing Racer（惊人的赛车）游戏，它要求你运用迄今为止学到的所有知识。

第**8**章，**“**编写第**1**部分的脚本**”**：在本章中，你将开始尝试利用Unity 编写脚本。如果你以前从未编写过程序，也不要担心。我们将尽量放慢进度，以便你学习基础知识。

第**9**章，**“**编写第**2**部分的脚本**”**：在本章中，将扩展你在第8章中所学的知识。这一次，你将重点关注更高级的主题。

第**10**章，**“**碰撞**”**：本章将逐一介绍现代视频游戏中常见的多种碰撞交互。你将学习物理碰撞以及触发碰撞，还将学习创建物理材质，以便给对象添加一些变化。

第**11**章，**“**第**2**款游戏：**Chaos Ball”**：是时候开发另一款游戏了！在本章中，你将创建 Chaos Ball（混乱球）游戏。本章的章名可以说是名副其实，因为你将实现多种碰撞、物理材质和目标，也许还会把策略与突然移动的反应结合起来。

第**12**章，**“**预设**”**：预设（prefab）是创建可重用的游戏对象的一种优秀方式。在本章中，你将学习创建和修改预设，还将学习在脚本中构建它们。

第**13**章，**“**图形用户界面**”**：在本章中，你将学习如何在 Unity 中实

现图形用户界面（graphical user interface，GUI），还将学习多种组件，以及如何在 2D 界面上定位它们。

第**14**章，**“**角色控制器**”**：在本章中，你将学习如何创建自己的角色控制器，并通过构建你自己的自定义控制器来结束本章。

第**15**章，**“**第**3**款游戏：Captain Blaster”：这将是你创建的第3 款游戏！在本章中，你将创建Captain Blaster（爆破队长）游戏，它是一款制动火箭式的宇宙飞船射击游戏。

第**16**章，**“**粒子系统**”**：是时候学习粒子效果了。在本章中，你将尝试Unity传统的粒子系统及其全新的Shuriken粒子系统，还将学习如何创建冷却效果，并把它们应用于你的项目。

第**17**章，**“**动画**”**：在本章中，你将开始学习动画以及Unity遗留的动画系统，还将试验使用Asset Store 中的资源使模型栩栩如生。

第**18**章，**“**动画器**”**：本章全都用于介绍 Unity 新引入的 Mecanim 动画系统，你将学习重新映射模型绑定，并对它们应用通用的动画。

第**19**章，**“**第**4**款游戏：Gauntlet Runner”：第4款游戏被命名为Gauntlet Runner，这款游戏探索了一种滚动背景的新方式，以及如何实现动画控制器来构建复杂的混合式动画。

第**20**章，**“**音频**”**：本章让你通过音频添加一些重要的环境效果，你将学习2D和3D音频以及它们不同的属性。

第**21**章，**“**移动开发**”**：在本章中，你将学习为移动设备构建游戏，还将学习利用移动设备的内置加速计和多点触摸显示屏（multi-touch display）。

第**22**章，**“**游戏修改**”**：现在应该回过头来并再次检查你创建的游戏。这一次将修改它们，使之能够在移动设备上工作。你将开始看到哪些控制模式可以很好地转换为移动模式，哪些则不能。

第**23**章，**“**润色和部署**”**：现在应该学习如何添加多个场景，以及在场景之间保留数据。你还将学习部署设置以及玩游戏。

第**24**章，**“**结束语**”**：在本章中，你将回顾并总结学习Unity所走过的旅程。本章提供了关于你做了什么以及接下来将去往哪里的有用信息。

Unity引擎版本

本书使用的Unity引擎版本为4.1和4.2。对你来说，使用这两个版本的效果几乎完全相同，但是要注意一些视觉元素可能改变了位置。例如，在一些屏幕图像中，你可能注意到Terrain菜单项出现在 Unity 编辑器顶部的菜单栏中。但在版本 4.2 中，它移动了位置。涉及地形创建和管理的所有解释都进行了更新，以说明新的进程。之所以在这里指出这一点，是希望在两个版本看上去稍有不同时，你不会感到困惑。

感谢你阅读我为本书写的前言！我希望你享受本书，并从中学到大量的知识。祝你在学习Unity游戏引擎的旅程中好运连连！

第1章 Unity简介

在本章中你将学到：

怎样安装Unity；

怎样创建新项目或者打开现有的项目；

怎样使用Unity编辑器；

怎样在Unity Scene 视图内导航。

本章的意义在于让你准备好在 Unity 环境中大显身手。我们首先将探讨不同的 Unity 许可证，选择其中一种，然后安装它。一旦安装完成，你将学习如何创建新项目以及打开现有的项目。你将打开强大的Unity 编辑器，并研究它的多种组件。最后，你将学习使用鼠标控制和键盘命令导航场景。本章打算让你亲自动手实践，因此在阅读时要下载Unity，并遵循相应的学习指导。

1.1 安装Unity

要开始使用 Unity，首先需要下载并安装它。如今，软件安装是一个十分简单、直观的过程，Unity也不例外。不过，在我们可以安装任何部分之前，需要探讨两个可用的Unity许可证：Unity Free 和Unity Pro。Unity Free 足以完成本书中的所有示例和项目。事实上，Unity Free包含制作商业游戏所需的一切功能。如果你想要使用更强大的功能（并且愿意花费金钱）， Unity Pro 提供了一套扩展的工具，可以给你提供真正“高价的”游戏引擎体验。如果对Unity Pro 感到好奇，但是不想购买它，Unity Free 提供了为期30 天的Unity Pro 免费试用许可证。你可以随意使用Unity Pro 的特性，并且确定它是否适合你。在访问Unity网站时，你可能还会注意到还有Android和iOS插件许可证。从Unity的最新版起，基本的移动插件就是免费的，并且Unity附带有它们。

1.1.1 下载并安装Unity

出于本章的目的，我们将假定你遵循的是Unity Free 许可证。如果你使用的是Unity Pro版本，过程将非常相似，唯一的变化是在选择许可证时。当你准备好开始下载并安装 Unity时，可以遵循下面这些步骤。

（1）从http://unity3d.com/unity/download/上的Unity3D下载页面下载Unity安装程序。

（2）运行安装程序并遵循提示进行安装，就像安装任何其他的软件一样。

（3）在提示时，确保保持选中Example Project、Unity DevelopmentWeb Player和MonoDevelop这些复选框，如图1.1所示。

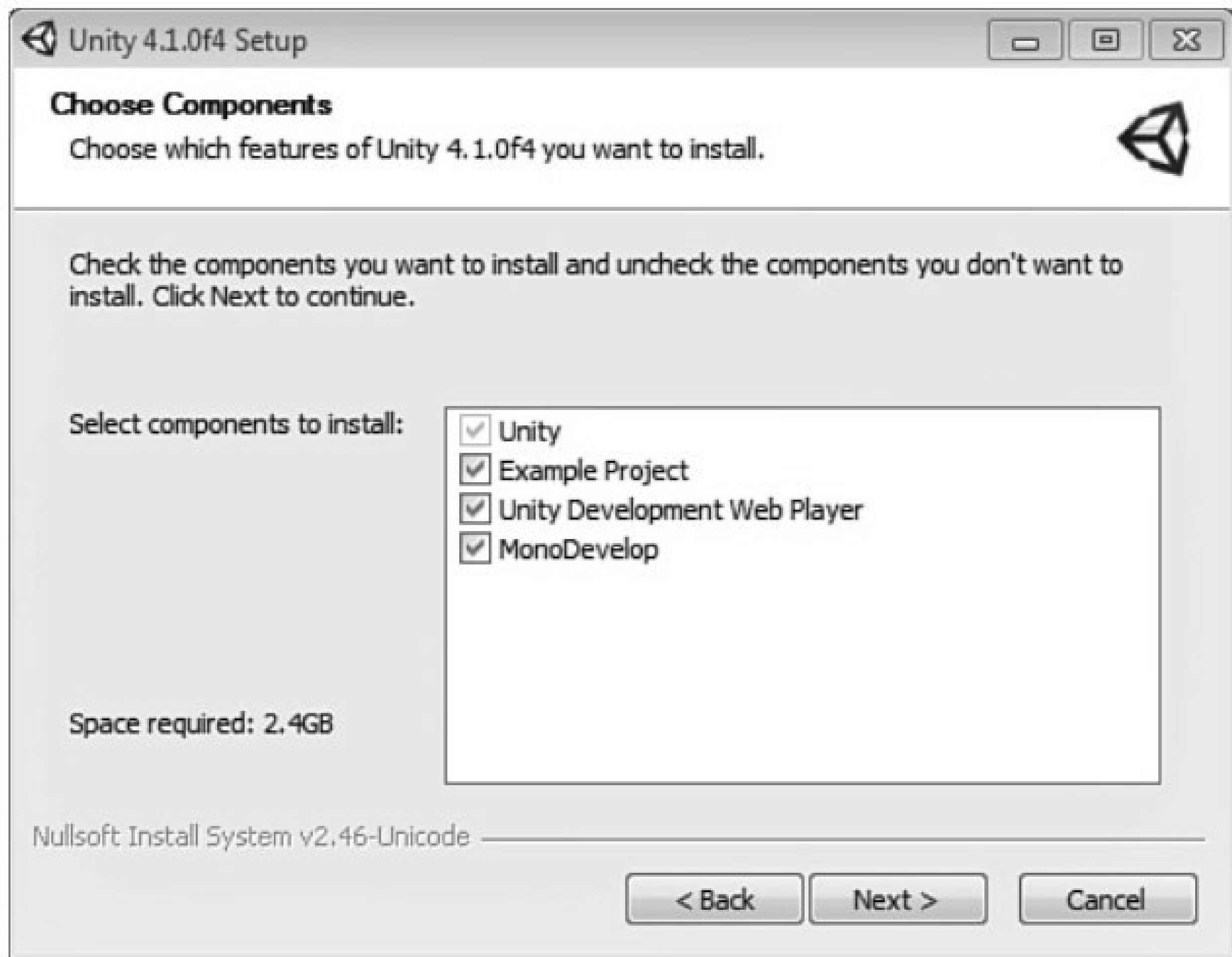


图1.1 提示选择要安装的组件

（4）为 Unity 选择一个安装位置，如图 1.2 所示。建议保留默认的设置，除非你确认自己这么做不会出问题。

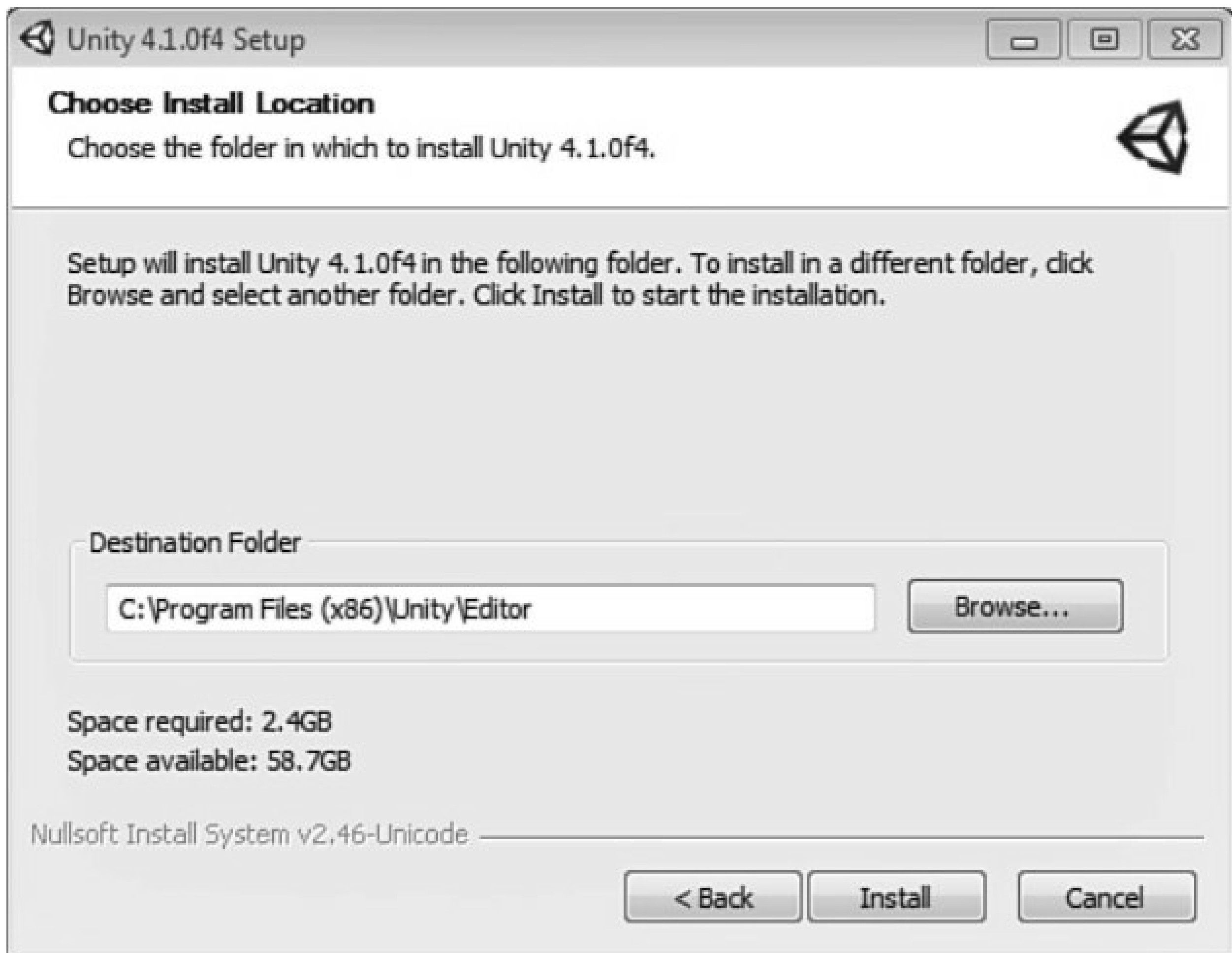


图1.2 提示选择安装位置

（5）至此，安装就将要完成了。

（6）在第一次运行 Unity 时，将要求你激活许可证，如图 1.3 所示。此时，可以选择你是想使用Unity Free，还是开始为期30 天的Unity Pro 免费试用。如果你购买了Unity Pro，可以输入序列号以解锁它。我们目前假定你选择的是Unity Free。

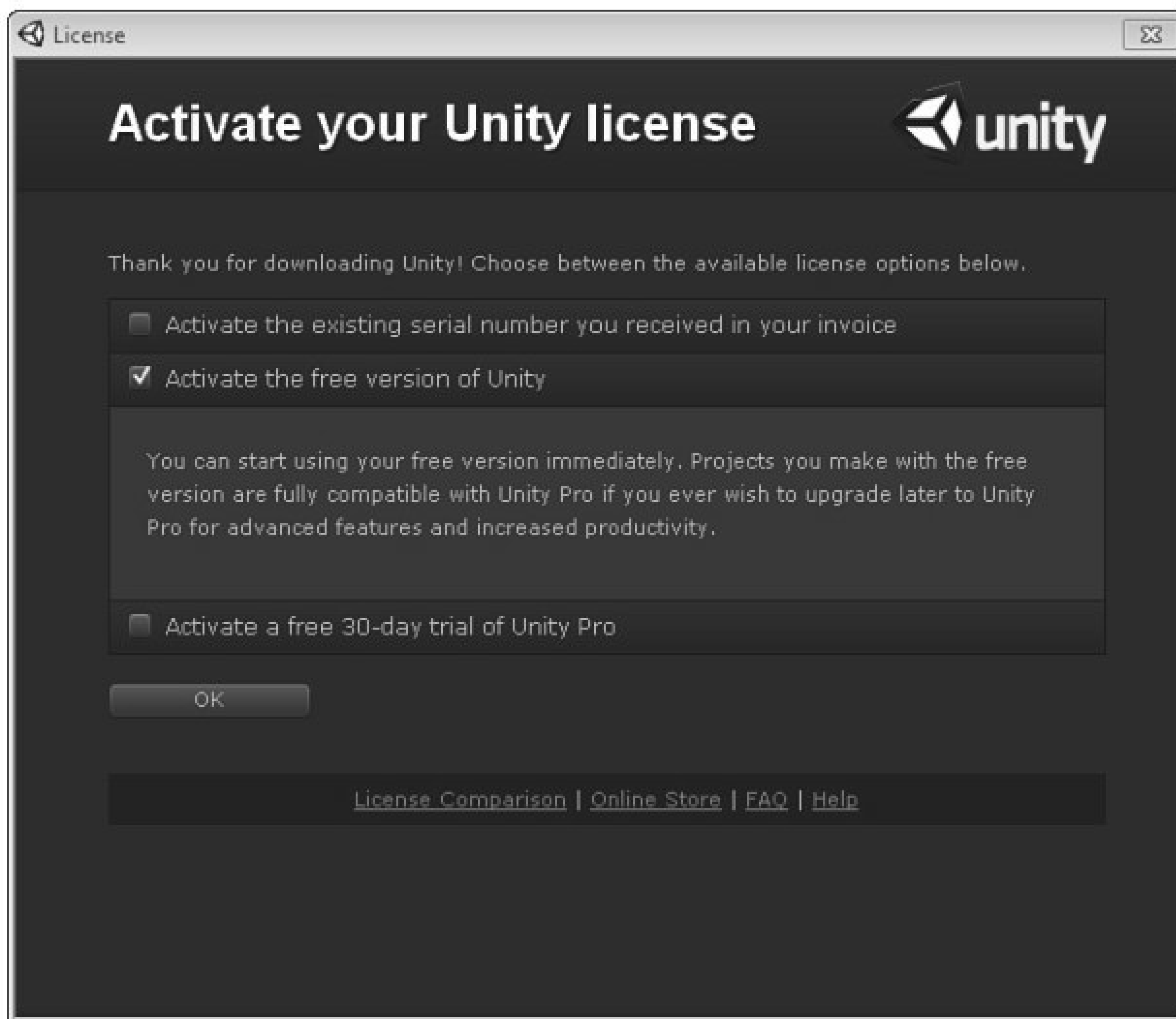


图1.3 Unity 许可证选择屏幕

（7）提示登录到一个Unity账户，如图1.4所示。如果具有一个Unity账户，可以在此输入它。如果没有，可以选择Create Account 选项，并填写必需的表单。

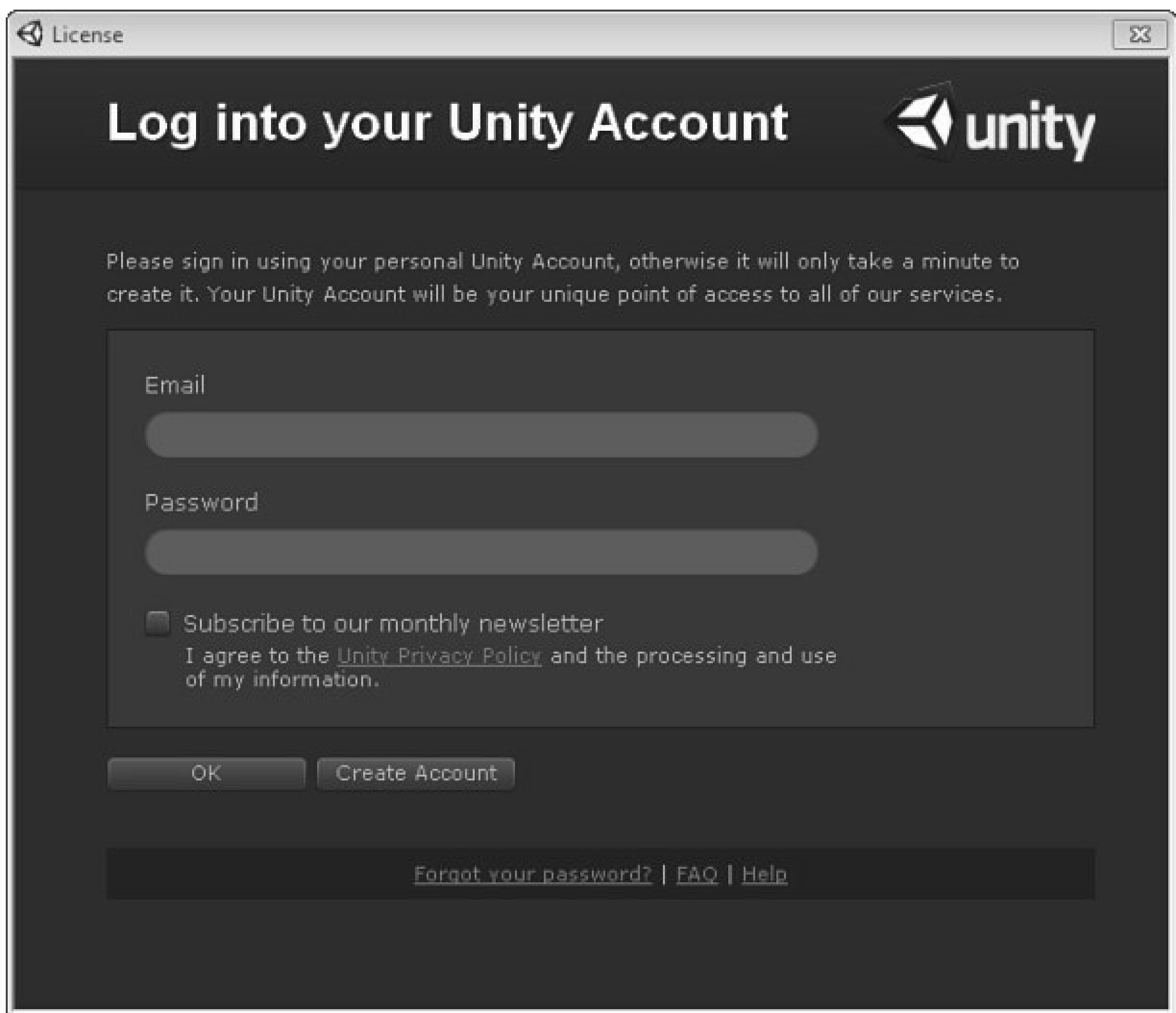


图1.4 提示登录到一个Unity账户

（8）这就行了！Unity安装现在就完成了。

注意：

支持的操作系统和硬件

要使用 Unity，必须使用 Windows PC 或 Macintosh 计算机。尽管可以构建在 Linux 系统上运行的项目，但是 Unity 编辑器本身却做不到这一点。你的计算机还必须满足以下最低要求（取自编写本书时的 Unity 网站上的内容）。

Windows：XP SP2 或更高版本。Mac OS X：Intel CPU 及 Snow Leopard 10.6 或更高版本。注意：Unity 没有在 Windows 和 OS X的服务

器版本上测试过。

具有 DirectX 9（Shader Model 2.0）能力的显卡，从 2004 年起生产的任何显卡都适用。

使用遮挡剔除（occlusion culling）需要具有遮挡查询支持的 GPU（一些 Intel GPU 不支持它）。

注意：这些只是最低要求。

警告：

Internet链接

所有的 Internet 链接在编写本书时都是最新的。不过，Web 位置有时会改变。如果我给你的链接不再提供你正在寻找的材料，那么使用Internet搜索应该会发现你所寻找的内容。

1.2 开始认识Unity编辑器

既然你已经安装了Unity，就可以开始探索Unity编辑器了。Unity编辑器是一种可视化组件，允许以一种“所见即所得”的方式构建游戏。由于大多数交互实际上都发生在我们与编辑器之间，因此通常就把它简称为Unity。本章的下一部分将研究Unity编辑器的各种元素，以及它们如何密切配合来制作游戏。

1.2.1 Project对话框

第一次打开Unity时，看到的第一个窗口是Project对话框，如图1.5所示。我们将使用这个窗口打开最近的项目、浏览已经创建的项目或者开始新的项目。

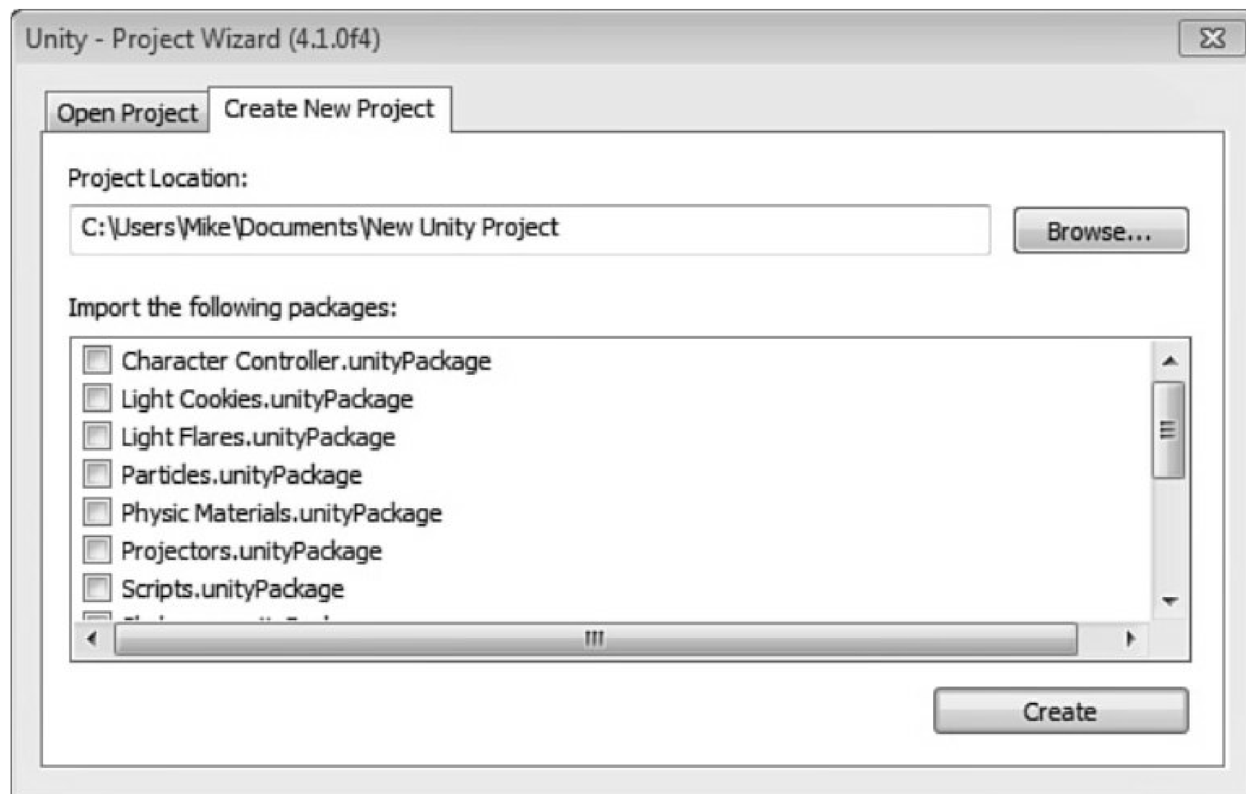


图1.5 Project对话框

如果已经在Unity中创建了一个项目，那么无论何时打开Unity，它都将直接进入该项目。要返回到Project 对话框，可以使用（从Unity里面）File > New Project 命令到达Create New Project 对话框，或者使用File > Open Project 命令到达Open Project 对话框。

提示：

打开Project对话框

在运行 Unity时，将会自动打开你处理的最后一个项目。如果你想打开Project 对话框而不是最后一个项目，可以按住 Alt 键（在 Mac 上则是Control键），并单击Unity图标。如果想要Unity总是保持这种工作方式，可以使用 Edit > Preferences 命令，并选中 Always Show Project Wizard复选框。

创建我们的第一个项目

现在让我们继续前进，并创建一个项目。你想知道在哪里保存项目，以便以后需要时可以轻松地找到它。如图1.6所示，显示了在创建项目之前的对话框的样子。

（1）打开Create New Project 对话框。

（2）选择一个保存项目的位置。如果你不知道该把项目存放在哪里，可以保留默认的选项。如果决定选择一个自定义的位置，可以选择一个空文件夹用于存放项目。这个空文件夹将命名为项目的名称。

（3）把项目命名为 Chapter1_Trial。项目名称是 Project Location 文本框中的最后一点文本。

（4）保持Import the Following Packages下的所有程序包都未选中，我们将在后面讨论程序包。

（5）单击Create按钮。

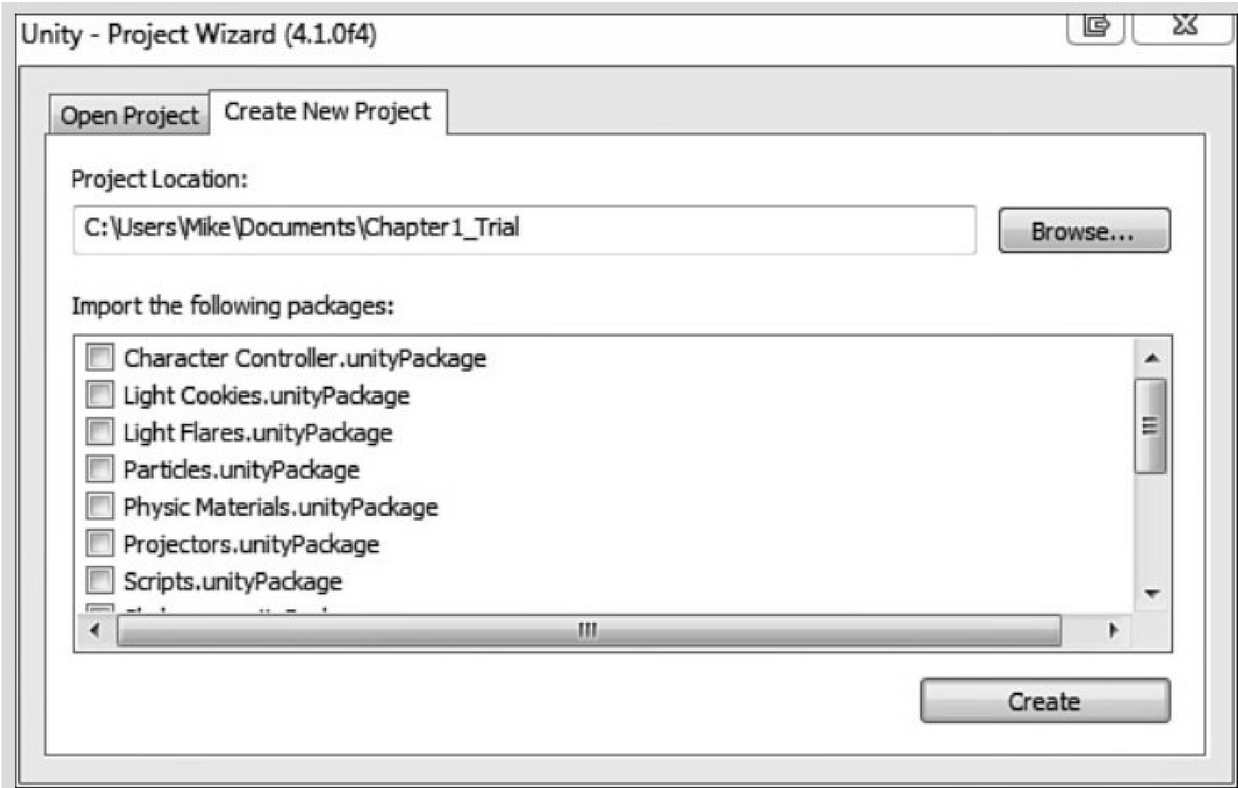


图1.6 用于第一个项目的设置

警告：

项目和程序包

起初，你可能忍不住在Create New Project对话框中选择一串程序包。不过，我想警告你的是，不要轻易地在项目中添加程序包，因为不需要的项目可能会增加文件的大小并延缓进度。未使用的程序包只会占据空间，而不会提供任何益处。记住了这一点，就知道更好的做法是：等到确实需要某个程序包时才导入它。即便如此，也只需要导入那个程序包中你打算使用的部分。

1.2.2 Unity界面

迄今为止，我们安装了Unity，并且查看了Project对话框。现在应该深入一步，开始尝试执行一些操作。在第一次打开一个新的 Unity 项目时，将会看到一堆灰色窗口，称为视图（view），并且所有的内容都在

一定程度上是灰色的，如图1.7所示。无需恐惧，我们将迅速使这个位置变得鲜活起来。在下面几节中，我们将逐一探讨每个独特的视图。不过，现在我想作为一个整体讨论Unity界面的布局。

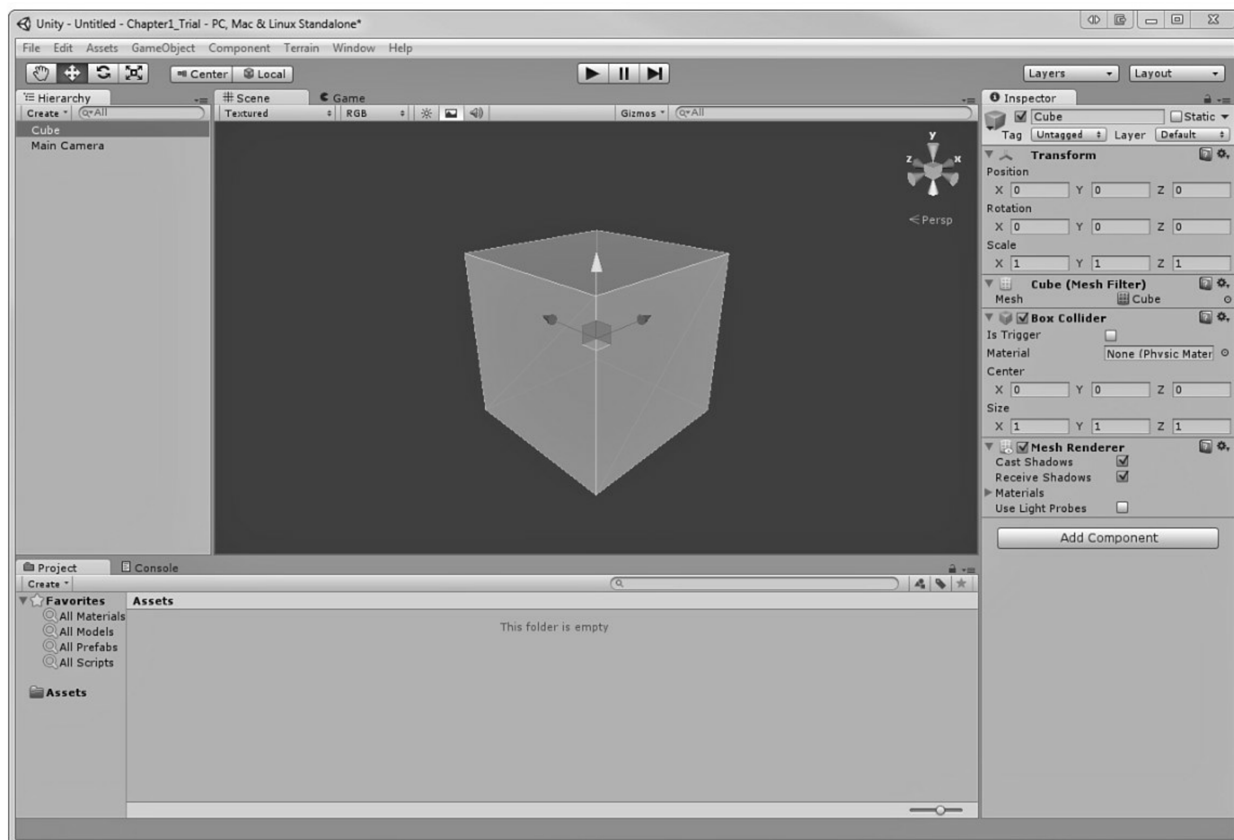


图1.7 Unity界面

对于初学者，Unity 允许用户定制他们的工作方式。这意味着任何视图都可以移动、停靠、复制或改变。例如，如果单击单词“Hierarchy”（位于左边）以选取Hierarchy视图，并把它拖到 Inspector上（位于右边），就可以把这两个视图叠加在一起。也可以把光标放在视图之间的任何线条上，并调整窗口大小。事实上，为什么不花点时间尝试移动一些内容，以使它们像你所喜欢的那样排列呢？如果最终得到的布局不是你喜欢的样子，也不要感到恐惧。可以使用Window > Layouts > Default Layout命令，快速、轻松地切换回内置的默认视图。虽然我们讨论的是内置布局的主题，但是可以更进一步，并试验几种其

他的布局（我是Wide布局的“粉丝”）。如果你创建了一种自己喜欢的布局，总是可以使用Window > Layouts > Save Layout命令把它保存起来。并且，即使你不小心改变了布局，也可以恢复它。

注意：

找到合适的布局

没有哪两个人是相同的，同样，没有哪两个人喜欢的布局是相同的。良好的布局有助于处理项目，并使工作容易得多。一定要花时间摆弄布局，找到一种最适合你的布局。你将利用 Unity做许多工作，以一种舒适的方式建立环境是有好处的。

如果你想要复制一种视图，它也是一个相当直观的过程。可以简单地右键单击任何视图标签（标签[tab]是显示视图名称的部分），把鼠标光标悬停在Add Tab 上，将会弹出一个视图列表，可供你从中选择，如图 1.8 所示。你可能想知道自己为什么想要复制视图。在疯狂地移动视图时，有可能意外地关闭视图。重新添加标签将会恢复它。此外，考虑创建多个Scene视图的能力。每个Scene视图都可以与项目内的特定元素或轴对齐。如果希望看到它的实际应用，可以使用Window > Layouts > 4 Split 命令，检查4种Split 内置布局（如果创建了一种你喜欢的布局，就一定要先保存它）。

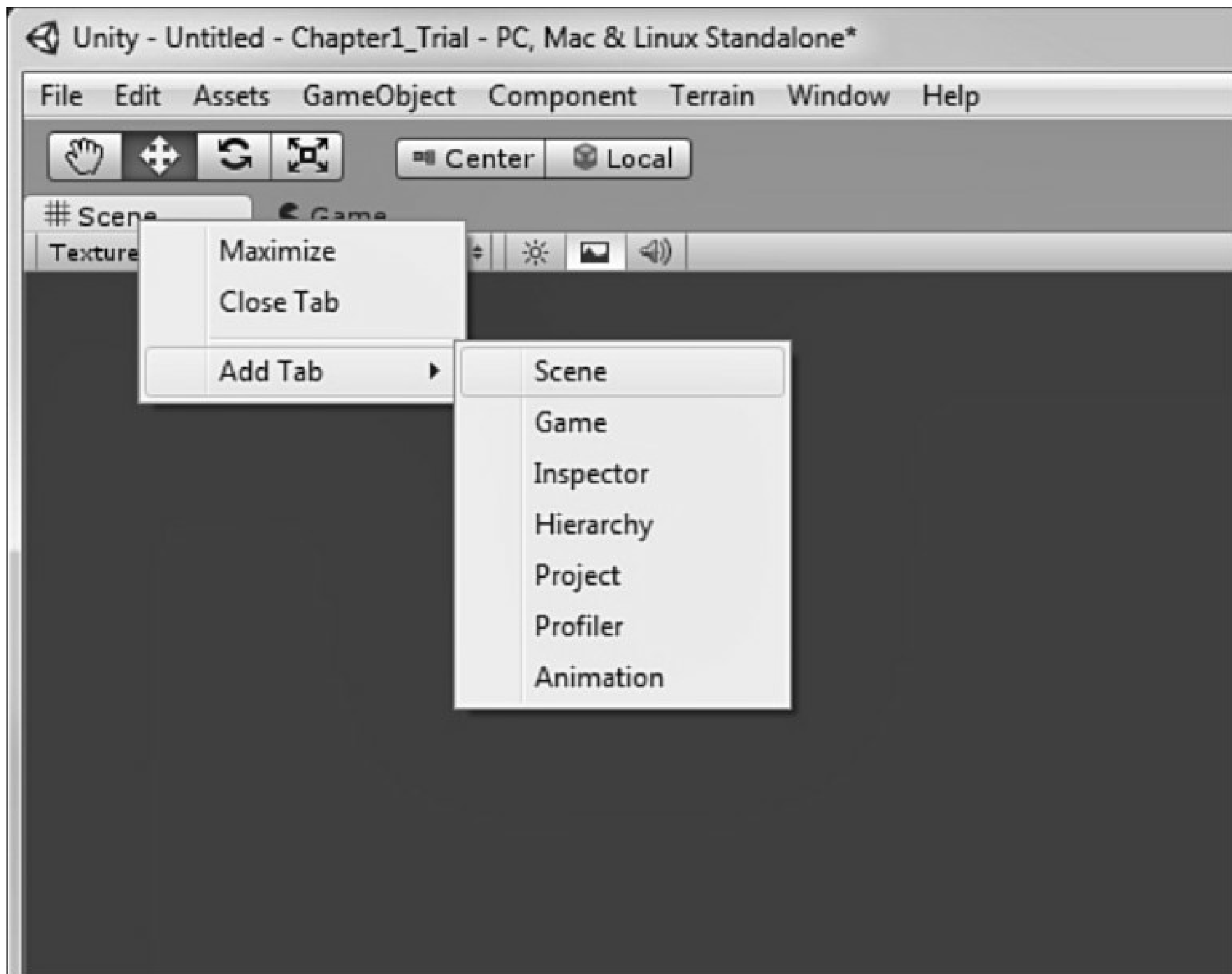


图1.8 添加新标签

现在，没有更多的骚扰了，让我们查看特定的视图本身。

1.2.3 Project视图

为一个项目创建的所有内容（文件、脚本、纹理、模型等）都可以在Project视图中找到，如图 1.9 所示。在这个窗口中，显示了整个项目的所有资源和组织结构。在创建一个新项目时，将会注意到单个名为Assets的文件夹项。如果进入硬盘驱动器上保存项目的文件夹，也会发现Assets文件夹。这是由于Unity利用硬盘驱动器上的文件夹镜像了Project视图。如果在 Unity 中创建一个文件或文件夹，在资源管理器中将会出现对应的文件或文件夹（反之亦然）。可以通过简单的拖放操

作，在Project视图中移动项目。这使你能够在文件夹内放置项目，或者自由地重新组织项目。

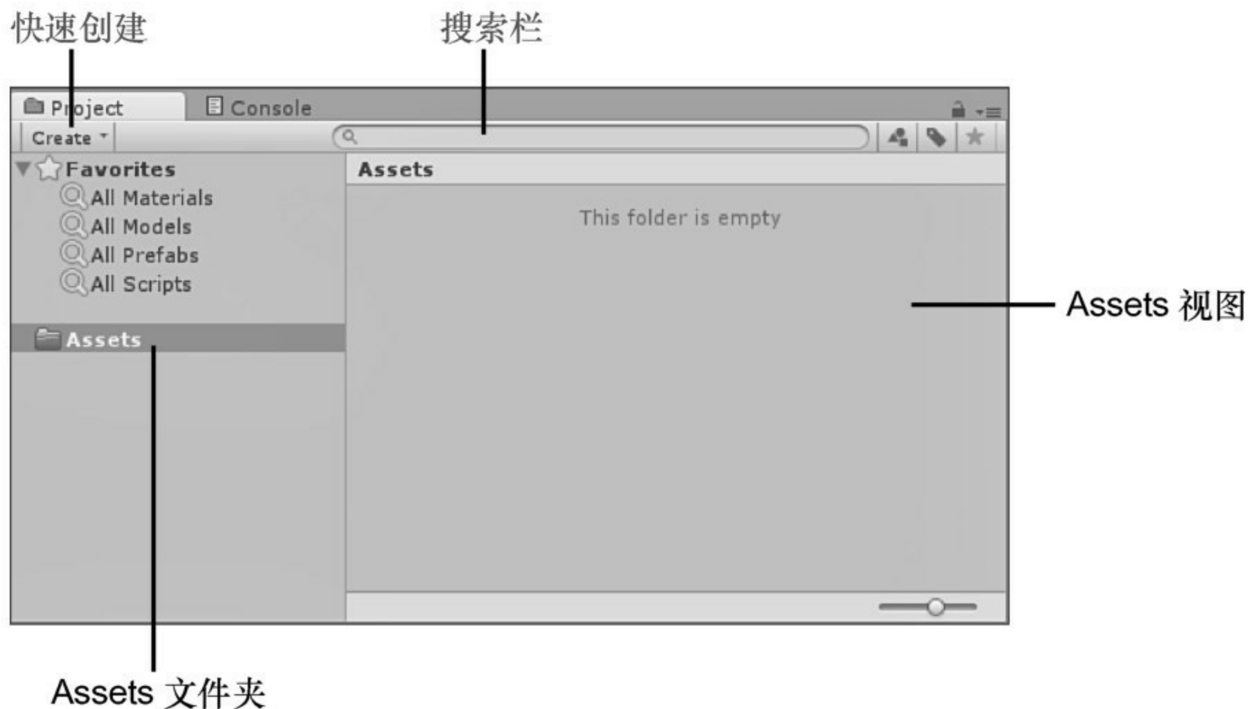


图1.9 Project视图

注意：

资源

资源是在资源文件夹中作为文件存在的任何项目，所有的纹理、网格、声音文件、脚本等都可以被视为资源。相反，如果创建一个游戏对象，但它没有创建对应的文件，那它就不是资源。

警告：

移动资源

Unity将维护与项目关联的多种资源之间的链接。因此，在Unity外面移动或删除项目可能会导致潜在的问题。一般来讲，在 Unity里面执行所有的资源管理是一个好主意。

无论何时，在Project视图中单击一个文件夹，都会在右边的Assets区域下面显示该文件夹的内容。如图 1.9 所示，Assets 文件夹目前是空

的，因此在右边不会显示任何内容。如果想要创建资源，可以通过单击Create下拉菜单，轻松做到这一点。这个菜单使你能够把各种资源和文件夹添加到项目中。

提示：

项目组织

项目组织对项目管理极其重要。随着项目变大，资源的数量也将开始增多，直至查找任何内容都可能变成一件繁重的工作。利用一些简单的组织规则，就可以避免许多问题的产生。

每种资源类型（场景、脚本、纹理等）都应该具有它自己的文件夹。

每种资源都应该位于一个文件夹中。

如果将要在一个文件夹内使用另一个文件夹，就要确保结构是有意义的。文件夹应该变得更具体，而不是含混不清。

遵循这几个简单的规则，便可以带来不同的结果。

在 Unity 4 中，添加到 Project 视图中的我最喜爱的特性之一是：添加了 Unity Asset Store 的收藏夹和集成。Favorites 按钮使你能够快速选择某种类型的所有资源，这使得你有可能快速获得资源的“一目了然”的视图。当单击 Favorites 按钮之一（例如，All Models）或者利用内置的搜索栏执行搜索时，将会看到在 Assets 与 Asset Store 之间可以缩小结果的范围。如果单击 Asset Store，将能够从 Unity Asset Store 中浏览适合搜索条件的资源，如图1.10所示。可以按免费资源和付费资源进一步缩小结果的范围。对我而言，这相当于添加了一个极佳的功能，因为它使你无需离开 Unity界面，就能够抓取项目所需的资源。



图1.10 搜索Unity Asset Store

1.2.4 Hierarchy视图

如图1.11所示，Hierarchy视图在许多方面都比较像Project视图。它们之间的区别是：Hierarchy视图显示了当前场景而不是整个项目（project）中的所有分项（item）。在第一次利用Unity 创建项目时，将会获得默认的场景，除了单个分项即Main Camera 之外，其他都是空的。在向场景中添加分项时，它们将出现在Hierarchy视图中。就像Project视图一样，可以使用Create菜单向场景中快速添加分项，使用内置的搜索栏执行搜索，以及单击并拖动项目来组织和“嵌套”它们。

嵌套

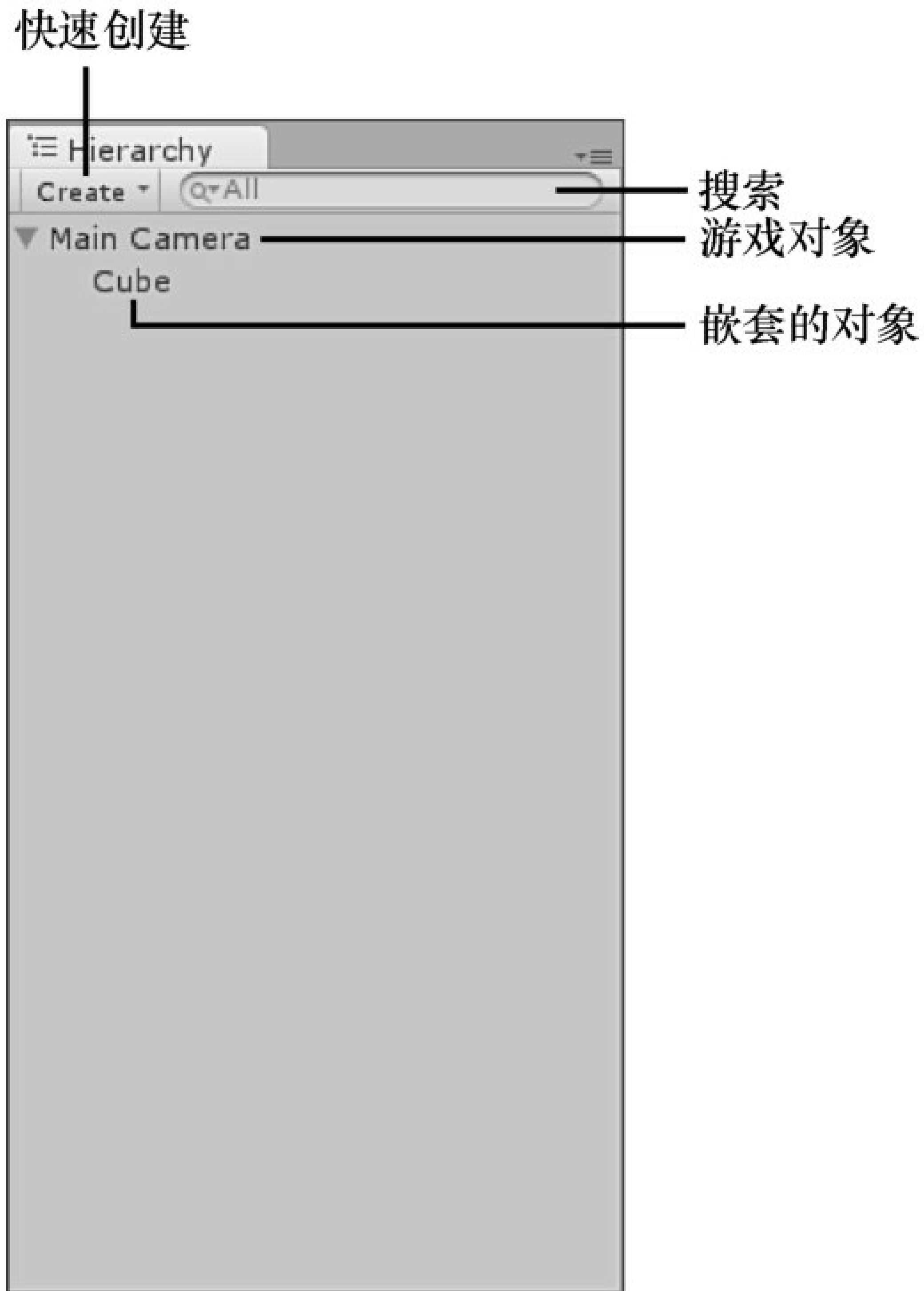


图1.11 Hierarchy视图

提示：

嵌套

嵌套（nesting）是用于在两个或更多的分项之间建立关系的术语。

在Hierarchy视图中，单击一个分项并把它拖到另一个分项上，将把这个分项嵌套在另一个分项之下，这通常被称为父/子关系。在这种情况下，顶部的对象是父对象，其下的任何对象都是子对象。你将知道一个对象是嵌套于另一个对象之下的，因为它以缩进形式显示。后面将会看到，在Hierarchy视图中嵌套对象可能会影响它们的行为方式。

提示：

场景

Unity使用场景（Scene）这个术语来描述你可能已经知道的作为一个关卡的内容。在开发 Unity项目时，对象和行为的每个集合都应该是它自己的场景。因此，如果正在利用雪关卡和丛林关卡构建游戏，它们将是单独的场景。

提示：

场景组织

在处理新的Unity项目时，应该做的第一件事是在Project视图中的Assets下面创建一个Scenes文件夹。这样，将把所有的场景（或关卡）存储在相同的位置。一定要给场景提供一个描述性的名称。Scene1现在听起来像是一个非常好的名称，但是当具有 30 个场景时，它可能会令人混淆。

1.2.5 Inspector视图

Inspector视图使你能够查看当前所选项目的所有属性。从Project或Hierarchy视图中简单地单击任何资源或对象，Inspector视图将自动传播相关的信息。

如图1.12所示，可以查看在Hierarchy视图中选择了Main Camera对象之后的Inspector视图。

让我们分解它的一些功能。

如果单击对象名称旁边的复选框，就会禁用它，并且它将不会出现在项目中。

下拉列表（比如 Layer 或 Tag 列表，后面将介绍关于它们的更多知识）用于从一组预先定义的选项中做出选择。

文本框、下拉菜单和滑块可以更改它们的值，并且所做的更改将自动、立即反映在场景中——即使游戏正在运行亦会如此！

每个游戏对象都充当不同组件（比如图1.12所示的Transform、Camera和GUILayer）。可以通过取消选中这些组件或者右键单击并选择Remove Component命令移除它们，来禁用这些组件。

可以单击Add Component按钮添加组件。

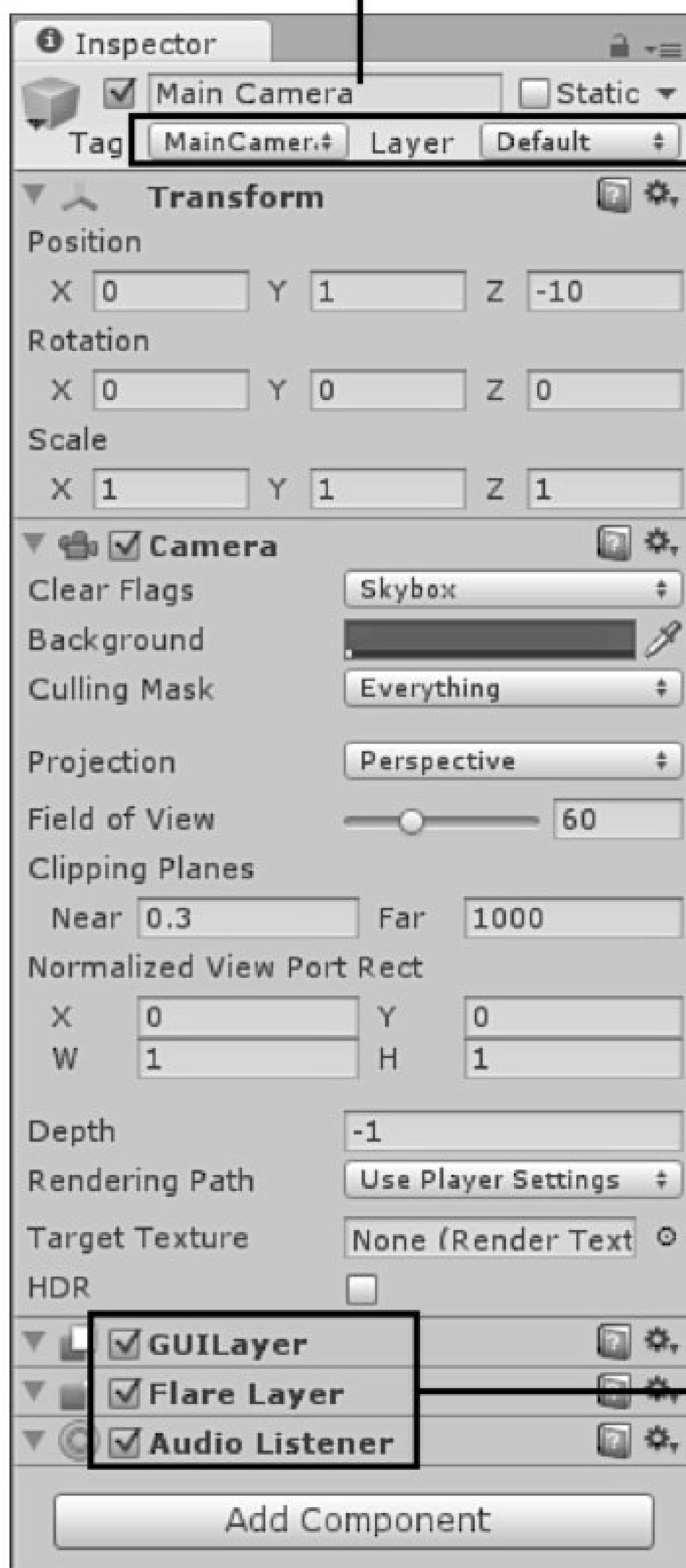

名称
下拉菜单
组件
Inspector
Main Camera
Static
Tag
MainCamer
Layer
Default
Transform
Position
X 0 Y 1 Z -10
Rotation
X 0 Y 0 Z 0
Scale
X 1 Y 1 Z 1
Camera
Clear Flags
Skybox
Background
Culling Mask
Everything
Projection
Perspective
Field of View
60
Clipping Planes
Near 0.3 Far 1000
Normalized View Port Rect
X 0 Y 0
W 1 H 1
Depth
-1
Rendering Path
Use Player Settings
Target Texture
None (Render Text
HDR
GUILayer
Flare Layer
Audio Listener
Add Component

图1.12 Inspector视图

警告：

在运行场景时更改属性

更改对象的属性并且查看那些立即反映在运行场景中的更改能力非常强大。它使你能够完全动态地调整诸如运动速度、跳跃高度、碰撞力度之类的属性，而无需停止和启动游戏。不过，要小心谨慎。在场景运行时对对象属性所做的任何更改都将在场景结束时更改回来。如果执行了更改并且喜欢所得到的结果，就一定要记住它是什么，以便在场景停止时可以再次设置它。

1.2.6 Scene视图

Scene视图是你将使用的最重要的视图，因为它使你能够在构建游戏时可视化地查看它，如图1.13所示。使用鼠标控制和几个热键，可以在场景内四处移动，并且在需要的地方放置对象。这给你提供了一种极佳的控制级别。

稍后，我们将讨论在场景内四处移动，但是，首先让我们重点关注作为Scene视图一部分的控件。

绘图模式：这用于控制如何描绘场景。默认把它设置为 Textured，这意味着将利用它们的纹理绘制对象。

渲染模式：这用于控制如何绘制场景中的对象。默认情况下，渲染模式是RGB，这意味着将以它们的全彩色（full color）绘制对象。

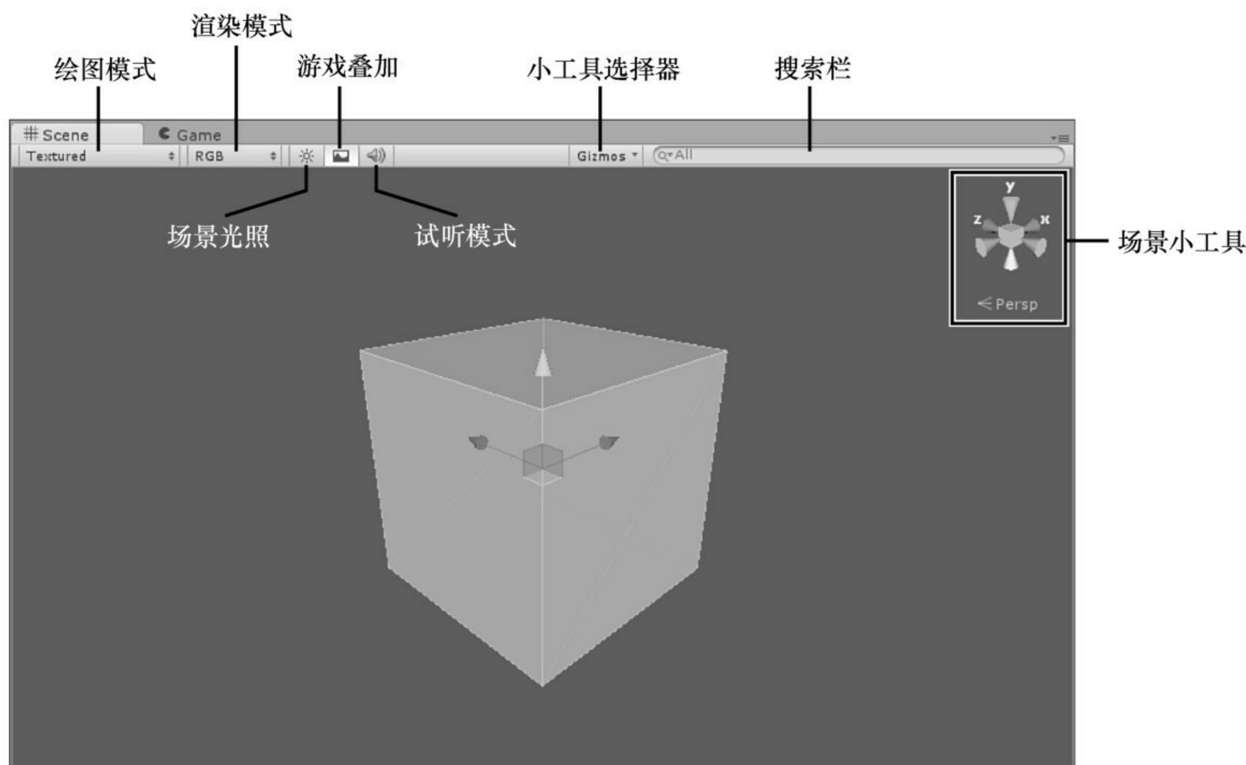


图1.13 Scene视图

场景光照：这个控件用于确定Scene视图中的对象是将通过默认的环境光照（ambient lighting）还是通过实际存在于场景内的灯光来照明。默认是使用内置的环境光照，但是一旦向场景中添加了第一个灯光，就会改变默认设置。

游戏叠加（overlay）：此控件用于确定像天空盒（skybox）和图形用户界面（GUI）这样的组件如何出现在Scene视图中，它还用于控制位置网格是否可见。

试听模式（audition mode）：这个控件用于设置 Scene 视图中的音频源是否正常工作。

小工具选择器（gizmo selector）：这个控件让你能够选择哪些“小工具”将出现在Scene视图中。小工具是指示器，可以提供可视化的调试或者辅助进行设置。

场景小工具（scene gizmo）：这个控件显示了你目前面对的是哪个方向，以及把Scene视图与轴对齐。

注意：

打开

场景小工具

场景小工具在Scene视图上提供了许多功能。可以看到，该控件具有与坐标轴对齐的X、Y和Z指示器。这使得很容易准确判断你在场景中观看的是哪个方向。在后面一章中将讨论轴和 3D 空间。小工具还允许主动控制场景的对齐方式。如果单击小工具的其中一根轴，将会注意到Scene视图立即贴紧那根轴，并且被放置到某个方向，比如顶部或左边。单击小工具中心的方框，将在Iso模式与Persp模式之间切换。Iso代表Isometric，它是没有应用透视的3D视图。与之相反，Persp代表Perspective，是应用了透视的3D视图。自己试试它，查看它是如何影响Scene视图的。

注意：

不同的版本，不同的按钮

如果使用Unity 4.2或之前的版本，场景视图菜单看上去如图1.13所示的那样。不过，如果使用Unity 4.3 或之后的版本，它们看上去将稍有不同。不要担心，选项仍然都在那里，它们现在只是位于 Effects 下拉菜单之下。你还可能注意到一个新的2D按钮，在前面的图像中没有显示出它。这个按钮支持使用Unity新增的2D能力。不过，由于本书重点关注的是3D游戏，目前还不需要关心这些选项。

1.2.7 Game视图

我们将要查看的最后一个视图是 Game 视图。实质上讲，Game 视图通过提供当前场景的完全模拟，允许在编辑器内“玩”游戏。游戏的所有元素都将在Game视图中正常工作，就像完全构建了项目一样。图1.14显示了Game视图的样子。注意：尽管从技术上讲Play、Pause和Step按

钮不是Game视图的一部分，但是它们可以控制Game视图，因此把它们包括在图像中。

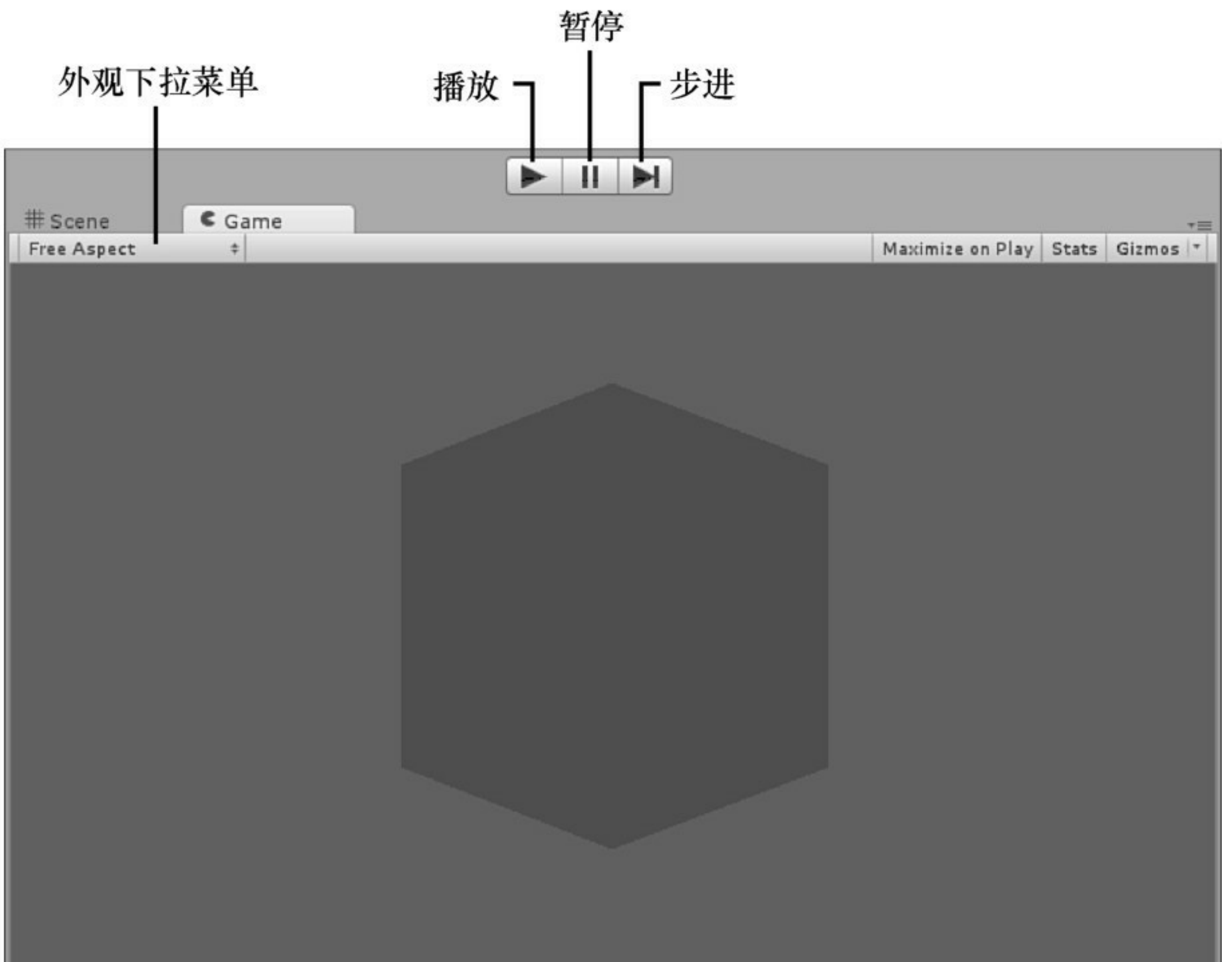


图1.14 Game视图

提示：

丢失的Game视图

如果发现Game视图隐藏在Scene视图后面，或者Game视图选项卡完全消失了，不要担心。一单击 Play 按钮，Game 视图选项卡就会出现在编辑器中，并且开始显示游戏。

Game视图带有一些控件，可以帮助我们测试游戏。

播放：Play按钮使你能够播放当前的场景。所有的控件、动画、声音和效果都将存在并且会工作。一旦游戏运行，就好像游戏是在独立的

播放器（比如在 PC 或移动设备上）中运行一样。要使游戏停止运行，可再次单击Play按钮。

暂停：Pause按钮用于暂停当前正在运行的Game 视图。游戏将维持它的状态，并且可以从暂停时它所处的位置继续运行。再次单击Pause按钮，将继续运行游戏。

步进：Step 按钮在 Game 视图暂停时可用，可以让游戏以单帧的速度运行。这实际上允许缓慢地“单步”通过游戏，并调试你可能具有的任何问题。当游戏正在运行时，按下Step按钮将导致游戏暂停。

外观下拉菜单：从这个下拉菜单中，可以选择你希望 Game 视图窗口在运行时所显示的高宽比。默认选项是Free Aspect，但是可以更改它，以匹配你为之开发游戏的目标平台的高宽比。

Maximize on Play：这个按钮用于确定Game 视图在运行时是否会占据编辑器的全部空间。它默认是关闭的，并且正在运行的游戏只会占据Game视图选项卡的大小。

Stats：这个按钮用于确定当游戏正在运行时是否在屏幕上显示渲染的统计信息。这些统计信息可用于度量场景的效率。默认将该按钮设置为关闭状态。

Gizmos：它既是一个按钮，也是一个下拉菜单。这个按钮用于确定当游戏正在运行时是否显示小工具，默认将其设置为关闭的。这个按钮上的下拉菜单（小箭头）用于确定：如果小工具是打开的，那么将显示哪个小工具。

注意：

运行、暂停和关闭

最初可能难以确定运行（running）、暂停（paused）和关闭（off）这些术语的含义。当没有在Game视图中执行游戏时，就称游戏是关闭的。当游戏处于关闭状态时，游戏控件将不会工作，并且不能播放游戏。当按下Play按钮时，游戏开始执行，就称游戏正在运行。播放、执

行和运行都意味着同一件事。如果游戏正在运行，并且按下Pause按钮，游戏就会停止运行，但是仍将维持它的状态。此时，游戏就处于暂停状态。暂停的游戏与关闭的游戏之间的区别是：暂停的游戏将从暂停它的位置恢复执行，而关闭的游戏则将从起始位置开始执行。

1.2.8 致敬：工具栏

工具栏尽管不是一个视图，但它是Unity编辑器的一个不必可少的部分。图1.15显示了工具栏的组件。

变换工具：这些按钮使你能够操纵游戏对象，将在后面更详细地介绍它们。要特别注意那个像人手一样的按钮，这是Hand工具，将在本章后面描述它。

变换小工具（gizmo）切换开关：这些切换开关可以操纵小工具将如何出现在Scene视图中，目前暂且不管它们。

Game 视图控件：这些按钮用于控制Game 视图。

图层下拉菜单：这个菜单用于确定哪些对象图层将出现在 Scene 视图中。默认情况下，所有的内容都将出现在Scene视图中。目前暂且不管它。在后面一章中将介绍图层。

布局下拉菜单：这个菜单允许快速更改编辑器的布局。

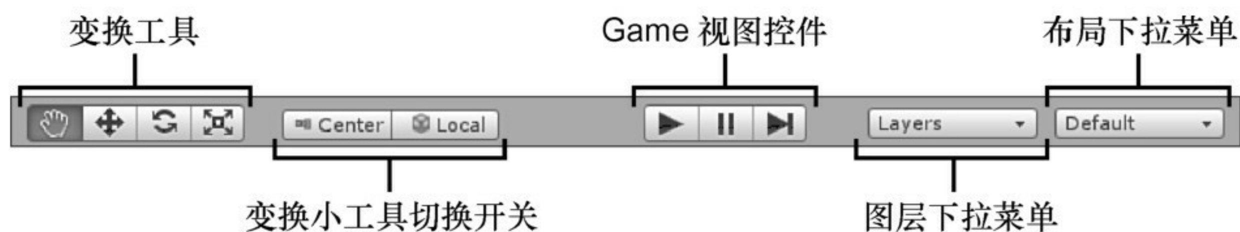


图1.15 工具栏

1.3 导航Unity 的Scene 视图

Scene视图允许对游戏的构造进行诸多控制。可视化地放置和修改项目的能力非常强大。不过，如果不能在场景内四处移动，所有这些将没有非常大的用处。本节将介绍两种不同的方式，用于更改位置和导航Scene视图。

1.3.1 Hand工具

Hand工具（热键：Q）提供了一种简单的机制，利用鼠标四处移动Scene视图，如图1.16所示。如果使用的是单键鼠标，那么这个工具被证明特别有用（因为其他方法都需要双键鼠标）。表1.1简要解释了Hand工具的每种控制方式。

表1.1 Hand工具的控制方式

动作	效果
单击并拖动	在场景中四处拖动摄像机
按住 Alt 键，然后单击并拖动	使摄像机围绕当前枢轴点转动
按住 Ctrl 键（在 Mac 上是 Command 键），然后右键单击并拖动	推近或拉远摄像机

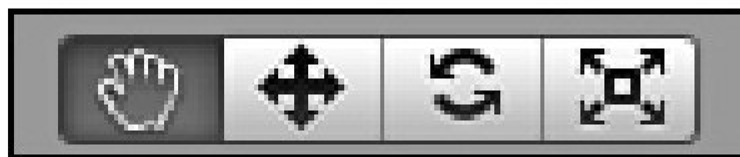


图1.16 Hand工具

在下面这个网址上可以找到所有的Unity热键：

http://blogs.unity3d.com/2011/08/24/unity-hotkeys-keyboard-shortcuts-in-unity/

警告：

不同的摄像机

在Unity中工作时，将与两类摄像机打交道。第一类是标准的游戏对象摄像机，可以看到在场景中已经具有这样的摄像机（默认情况）；第二类更多的是一种假想的摄像机，从传统意义上讲它不是一种摄像机，相反，它确定了我们在 Scene 视图中可以看到什么。在本章中，在提及摄像机时，所指的都是第二类摄像机。你将不会实际地操纵游戏对象摄像机。

1.3.2 Flythrough模式

Flythrough 模式使你能够使用传统的第一人称控制模式在场景中四处移动。对于任何玩第一人称 3D 游戏（比如第一人称射击游戏）的人来说，这种模式都让他们有一种宾至如归的感觉。如果你没有玩过那些游戏，可能要花点时间习惯这种模式。不过，一旦你熟悉了它，它就会变成一种习性。

按住鼠标右键将进入Flythrough模式。表1.2展示的所有动作都需要按住鼠标右键。

表1.2 Flythrough模式控制

动作	效果
移动鼠标	导致摄像机围绕枢轴转动，这提供了在场景内“四处查看”的感觉
按 W、A、S、D 键	W、A、S、D 键用于在场景内四处移动。其中每个键分别对应一个方向：前、左、后、右
按 Q、E 键	Q、E 键分别用于在场景内上、下移动
在按住 Shift 键的同时按下 W、A、S、D 键或 Q、E 键	与以前具有相同的效果，但它要快得多。可以把 Shift 键视作你的“全速移动”按钮

提示：

缩放

无论使用哪种方法进行导航，滚动鼠标滚轮总会在场景内缩放视图。默认情况下，场景将从Scene视图的中心进行放大或缩小。不过，如果在滚动时按住Alt 键，就会放大或缩小鼠标当前指向的任何内容。

继续前进并试验一下它！

提示：

贴紧控制

可以用多种方式精确控制场景导航。不过，有时，你只想在场景中四处转转。对于像这样的情况，使用所谓的贴紧控制（snap control）就很好。如果想要快速导航到或者放大场景中的某个游戏对象，可以高亮显示Hierarchy视图中的对象并按下F键。你将注意到场景“贴紧”到那个游戏对象。另一种贴紧控制是你已经见过的。场景小工具允许快速把摄像机贴紧到任何一根轴。这样，就可以从任意角度查看对象，而不必手动四处移动场景摄像机。一定要学习贴紧控制，并且利用它们快速导航场景就变成了一种贴紧操作！

1.4 小结

本章首先探讨了 Unity 游戏引擎，并通过下载和安装 Unity 开始这个过程。接着，你学习了如何打开和创建项目。然后，你学习了组成 Unity 编辑器的各种不同的视图，还学习了如何导航Scene视图。

1.5 问与答

问：资源和游戏对象是一样的吗？

答：不完全一样。基本上讲，它们之间最大的区别是：资源在硬盘驱动器上具有对应的文件或文件组，而游戏对象则不具有。资源可能会也可能不会包含游戏对象。

问：有许多不同的控件和选项，我需要立即把它们都记住吗？

答：根本不需要。大多数控件和选项都已经设置为涵盖了大多数情况的默认状态。随着你对 Unity 的认识进一步加深，可以继续学习关于可供你使用的不同控件的更多知识。本章只打算让你了解Unity编辑器中具有什么，并在一定程度上熟悉它们。

1.6 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

1.6.1 问题

1．判断题：你必须购买Unity Pro 才能制作游戏。

2．哪种视图使我们能够可视化地操纵场景中的对象？

3．判断题：总是应该在 Unity 内四处移动资源文件，而不应该使用操作系统的文件浏览器。

4．判断题：在创建一个新项目时，应该包括每种你认为极好的资源。

5．当按住鼠标右键时，在Scene视图中进入的是哪种模式？

1.6.2 答案

1．错误。

2．Scene视图。

3．正确。

4．错误。

5．Flythrough模式。

1.6.3 练习

花点时间练习一下本章中所学的概念。从根本上稳固理解 Unity 编辑器很重要，因为本书的一切知识都会以某种方式利用它。为了完成这个练习，可以执行以下操作。

1．使用File > New Scene命令或者按下Ctrl+N组合键（在Mac 上是Command+N组合键），创建一种新场景。

2．在Project视图中的Assets下面创建一个Scene文件夹。

3．使用File > Save Scene命令或者按下Ctrl+S组合键（在Mac上是Command+S组合键），保存场景。一定要在你创建的 Scenes 文件夹中保存场景，并给它提供一个描述性的名称。

4．向场景中添加一个立方体。为此，可以单击顶部的GameObject菜单，把鼠标放在Create Other 上，并从弹出式菜单中选择Cube 命令。

5．在Hierarchy视图中选择新添加的立方体，并在Inspector视图中试验它的属性。

6．练习使用Flythrough模式、Hand工具和贴紧控制在Scene视图中四处导航。使用立方体作为参照点，帮助你进行导航。

第2章 游戏对象

在本章中你将学到：

怎样处理2D 和3D 坐标；

怎样处理游戏对象；

怎样处理变换。

游戏对象是Unity游戏项目（project）的基本组件。场景中存在的每个分项（item）都是或者都基于游戏对象。在本章中，你将学习 Unity内的游戏对象。不过，在开始与 Unity 中的对象打交道之前，必须先学习2D和3D坐标系统。然后，将开始与内置的Unity游戏对象打交道，在本章最后，将学习多种不同的游戏对象变换。在本章中获得的信息对于本书其余各章是基础性的，一定要花时间学好它。

2.1 维度和坐标系统

华丽而富有魅力的视频游戏其实都是数学构造。所有的属性、运动和交互都可以归结为数字。对你来说幸运的是，许多地基都已经打好了。数学家们辛苦工作了几个世纪，发现、发明和简化了不同的过程，以便你可以利用现代的软件轻松地构建游戏。你可能认为游戏中的对象只是随意地存在于空间中，但是实际上每种游戏空间都具有维度，并且每个对象都放置在一种坐标系统（或网格）中。

2.1.1 在3D中放入一个维度

如前所述，每一款游戏都会使用某种级别的维度。你最有可能熟悉的最常见的维度系统是2D和3D，它们分别是二维（two-dimensional）和三维（three-dimensional）的简写形式。2D系统是一个平面系统。在2D系统中，只处理垂直和水平元素（或者换句话说是：上、下、左、右）。像Tetris、Pong和Pac　Man这样的游戏就是2D游戏的良好示例。3D 系统就像2D系统一样，但它显然多一个维度。在 3D 系统中，不仅具有水平和垂直方向（上、下、左、右），还具有深度（里和外）。图2.1在阐释2D正方形和3D正方形（另称为立方体，cube）方面做了很好的工作。注意在3D立方体中包括有深度轴，使之就像是“凸出”的一样。

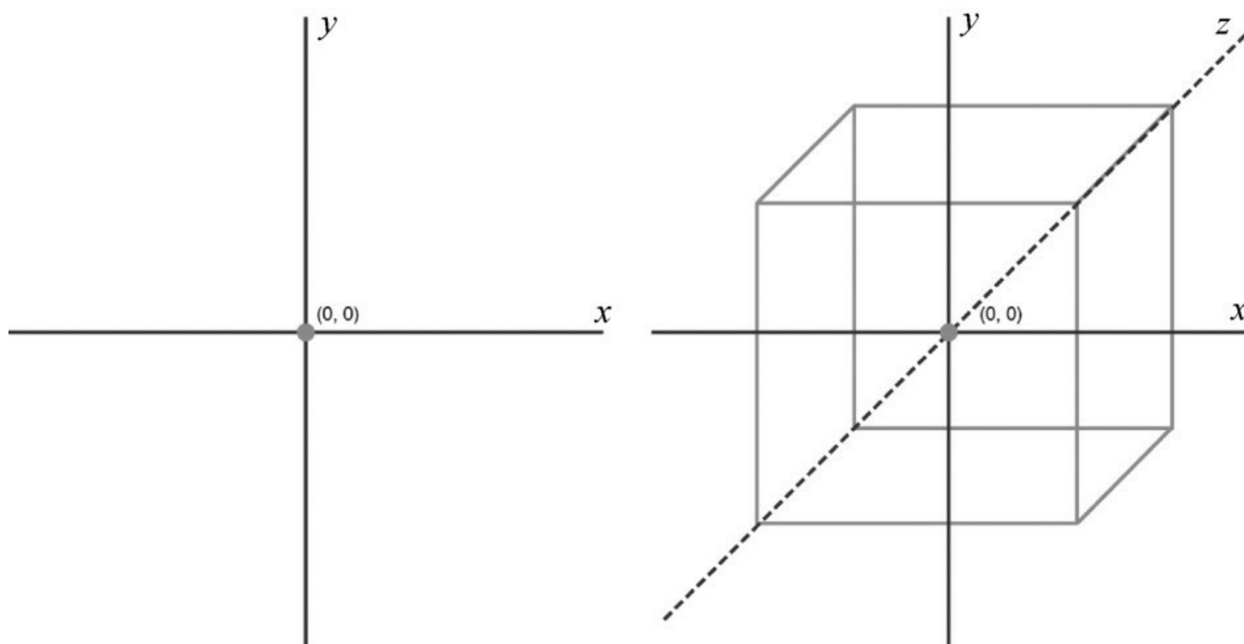


图2.1 2D正方形与3D立方体

注意：

学习2D和3D

Unity是一个3D引擎。因此，利用它创建的所有项目天生都使用三维系统。你可能想知道为什么我们要麻烦地完全涵盖 2D 系统。事实是：甚至在3D项目中，仍然有许多2D元素。纹理、屏幕元素和绘图技术都使用2D系统。学习2D系统是值得的，因为它们不会很快消失。

2.1.2 使用坐标系统

维度系统在数学上等价于坐标系统。坐标系统使用一系列直线（称为轴，axis）和位置（称为点，point），这些轴直接对应于它们模拟的维度。例如，2D坐标系统具有x轴和y轴，它们分别代表水平和垂直方向。如果在水平方向上移动一个对象，就称之为“沿着x轴”移动。同样，3D坐标系统使用x轴、y轴和z轴，分别代表水平、垂直和深度。

注意：

常用的坐标系统

在谈到对象的位置时，一般都会列出它的坐标。称对象在x轴上是2并且在y轴上是4可能有点麻烦。幸运的是，存在一种书写坐标的简写方式。在 2D 系统中，采用(x, y)这样的形式书写坐标。在 3D 系统中，则把它们写成(x, y, z)的形式。因此，将代之以把这个示例写作(2, 4)。如果那个对象还在z轴上是10，则将把它写作(2, 4, 10)。

每种坐标系统都具有所有的轴相交的点。这个点称为原点（origin），原点的坐标在 2D系统中总是(0, 0)，在3D系统中则是(0, 0, 0)。这个原点非常重要，因为它是得到其他所有点的基础。其他任何点的坐标都只是那个点沿着每根轴相距原点的距离。当移动某个点使之远离原点时，那个点的坐标将变大。例如，右移一个点时，它的x轴的值将变大；左移它时，x轴的值将变小，直到它经过原点为止。此时，点的x值将再次开始变大，但它也会变成负数。如图2.2 所示，这个2D坐标系统定义了3 个点。点(2, 2)在x方向上和y方向上都距离原点2个单位（unit）。点(−3, 3)位于原点的左边及其上方都是3 个单位。点(2, −2)位于原点的右边及其下方都是2个单位。

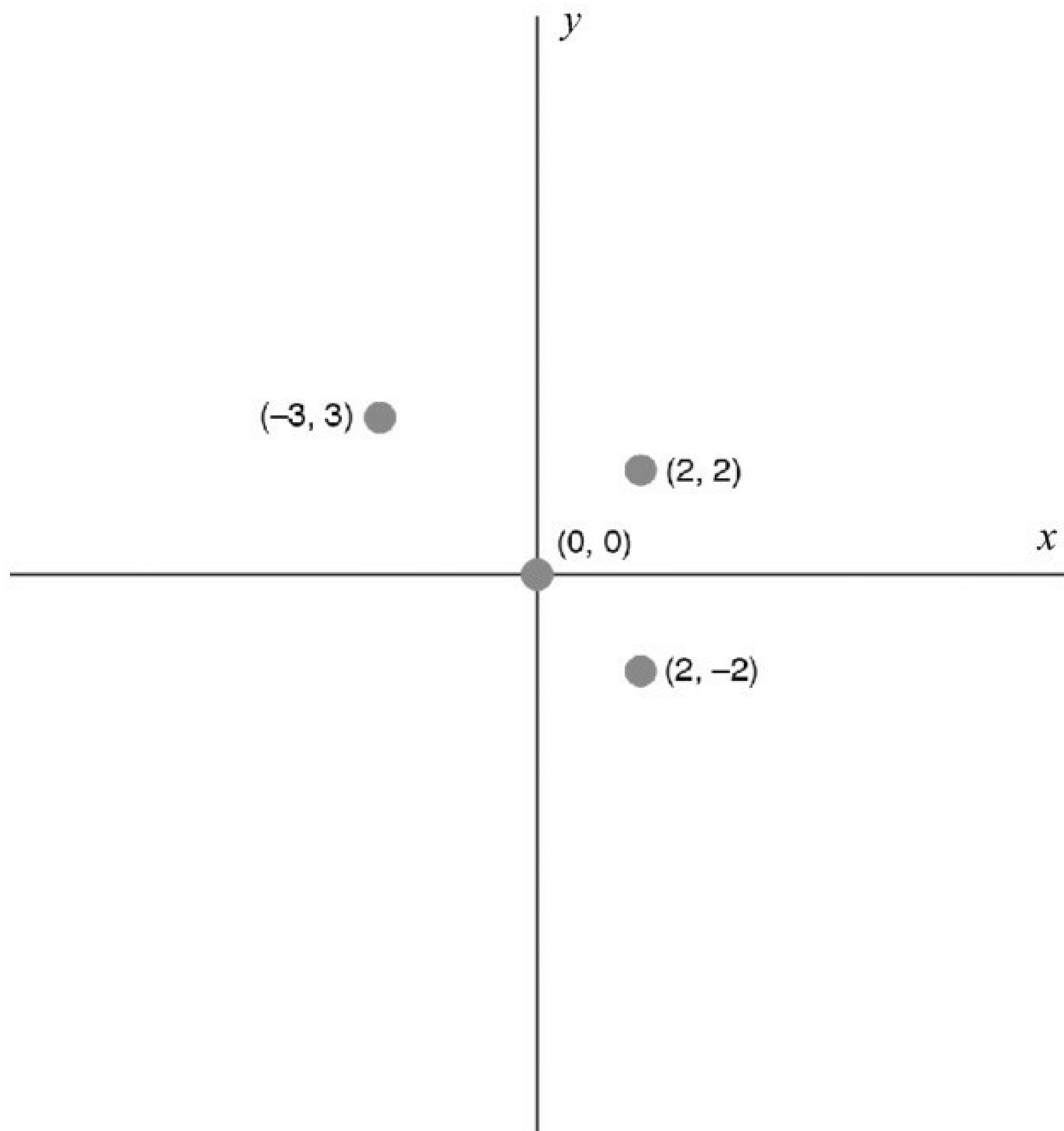


图2.2 点与原点的关系

2.1.3 世界坐标与局部坐标

你现在学习了游戏世界的维度以及组成它们的坐标系统。迄今为止使用的坐标系统被认为是世界（world）坐标系统。在任何给定的时

间，在世界坐标系统中都只有一根x轴、y轴和z轴。同样，所有的对象都共享唯一一个原点。你可能不知道的是，还有一种所谓的局部（local）坐标系统。这个系统对于每个对象都是独特的，并且它与其他对象完全分隔开。这个局部系统具有它自己的轴和原点，其他对象不会使用它们。图 2.3 通过为每种坐标系统显示构成一个正方形的4个点，阐释了世界坐标系统与局部坐标系统之间的区别。

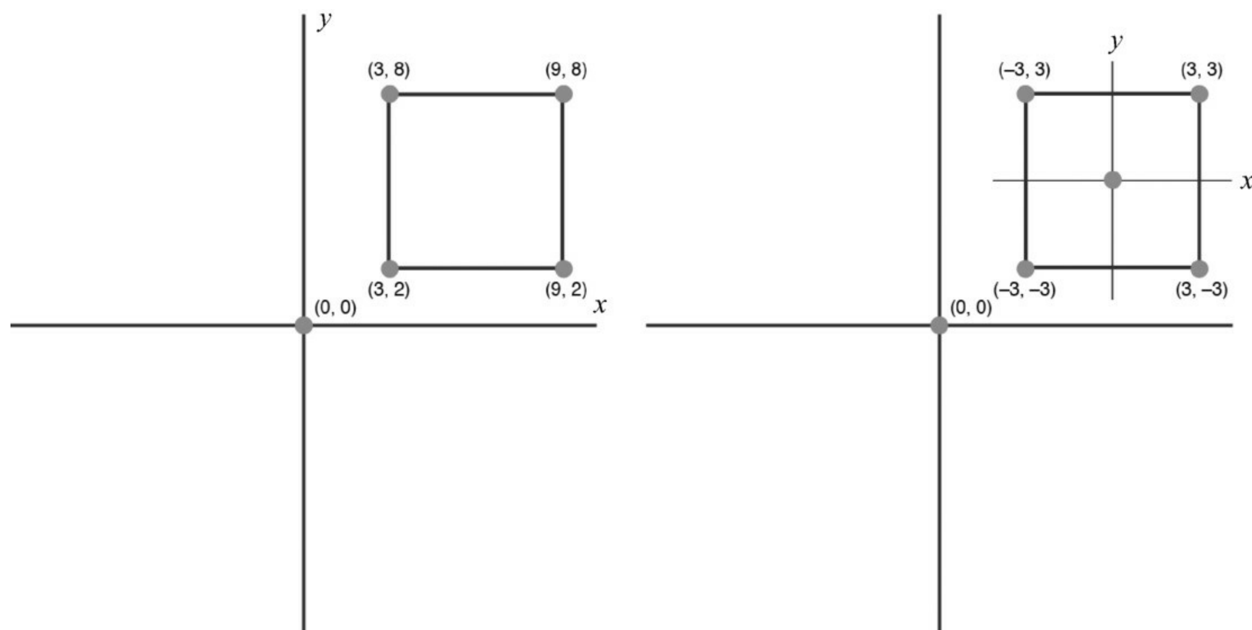


图2.3 世界坐标系统与局部坐标系统

你可能想问：如果世界坐标系统用于对象的位置，那么局部坐标系统是用于什么的呢？在本章后面，将探讨变换游戏对象以及设置游戏对象的父对象。它们都需要使用局部坐标系统。

2.2 游戏对象

Unity 游戏中的每种形状、模型、灯光、摄像机和粒子系统等都具有一个共同点：它们都是游戏对象。游戏对象是任何场景的基本单元。虽然它们比较简单，但是它们非常强大。归根究底，游戏对象差不多就是一种变换（将在本章后面更详细地讨论）和一个容器。这个容器用于保存多种使对象更动态、更有意义的组件。至于向游戏对象中添加什么，这取决于你自己。有许多组件，它们添加了大量的多样性。在整本书中，你将学习使用其中许多组件。

注意：

内置的对象

你使用的每个游戏对象在开始时并非都是一个空对象。Unity 具有多个内置的游戏对象可以开包即用。单击 Unity 编辑器顶部的GameObject菜单项，并把光标悬停在Create Other 上，就可以看到有大量的项目可用。学习使用Unity的很大一部分就是学习处理内置的和自定义的游戏对象。

创建一些游戏对象

现在让我们花一些时间处理游戏对象。你将创建几个基本的对象，并且检查它们的不同组件。

（1）创建一个新项目，或者在已经具有的项目中创建一个新场景。

（2）单击GameObject 菜单项并选择Create Empty 命令，添加一个空游戏对象（注意：也可以通过按下Ctrl+Shift+N组合键（PC用户）或者Command+Shift+N组合键（Mac用户），来创建空游戏对象）。

（3）查看Inspector视图，将会注意到你刚才创建的游戏对象除了

一种变换之外，将没有其他组件。所有的游戏对象都具有一种变换。在Inspector 视图中单击 Add Component按钮，将会显示可以添加到对象中的所有组件。此时不要选择任何组件。

（4）单击 GameObject 菜单项，把光标悬停在 Create Other 上，并从列表中选择 Cube命令，在项目中添加一个立方体。

（5）注意立方体具有空游戏对象所不具有的多种组年。网格（mesh）组件使立方体可见，碰撞器（collider）则使之能够与其他对象交互。

（6）最后，在Hierarchy 视图中单击Create 下拉菜单，并从列表中选择Point Light 命令，在项目中添加一个点光源（point light）。

（7）可以看到点光源没有与立方体共享任何组件，而是把注意力完全放在发光上。你还可能注意到：当把灯光添加到场景中时，其他对象将变暗。这是正常的，因为场景中存在灯光，所以Unity会关闭它的环境光照。

2.3 变换

至此，你学习并且探索了不同的坐标系统，并且试验了一些游戏对象。现在应该把它们二者融合起来。在处理3D对象时，经常会听到变换（transform）这个术语。依赖于上下文，变换要么是一个名词，要么是一个动词。3D空间里的所有对象都具有位置、旋转和比例。如果把它们都结合起来，就会得到对象的变换（名词）。此外，如果变换指的是更改对象的位置、旋转或比例，那它就可以是一个动词。Unity把它的两种含义与变换组件（transform component）结合起来，回忆可知变换组件是每个游戏对象都必须具有的唯一组件。每个空游戏对象都具有变换。使用这个组件，可以查看对象的当前变换以及改变（或变换）对象的变换。这现在听起来可能令人糊涂，但它相当简单，要不了多久你就会了解它。由于变换由位置、旋转和比例组成，改变变换就分别对应于平移、旋转和缩放。这些变换可以使用Inspector或变换工具实现。图2.4和图2.5显示了哪些Inspector组件和工具与哪些变换相关联。

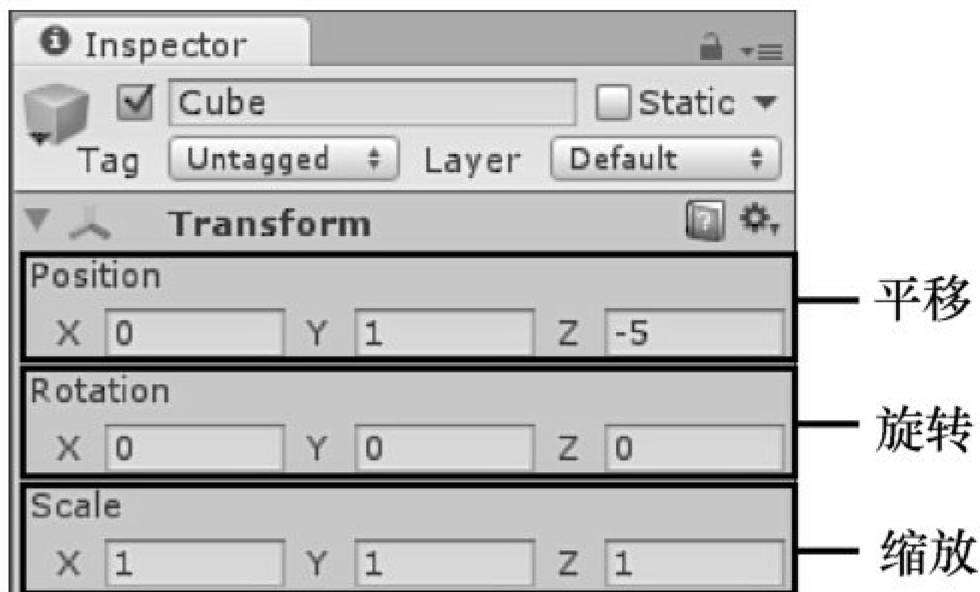


图2.4 Inspector 中的变换选项

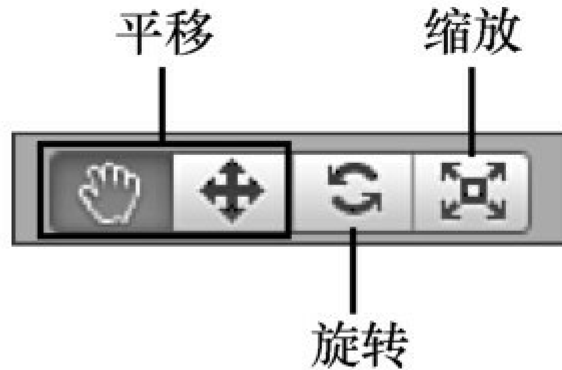


图2.5 变换工具

2.3.1 平移

在 3D 系统中改变对象的坐标位置称为平移（translation），这是可以应用于对象的最简单的变换。在平移一个对象时，将沿着轴移动它。图2.6演示了一个沿着x轴平移的正方形。

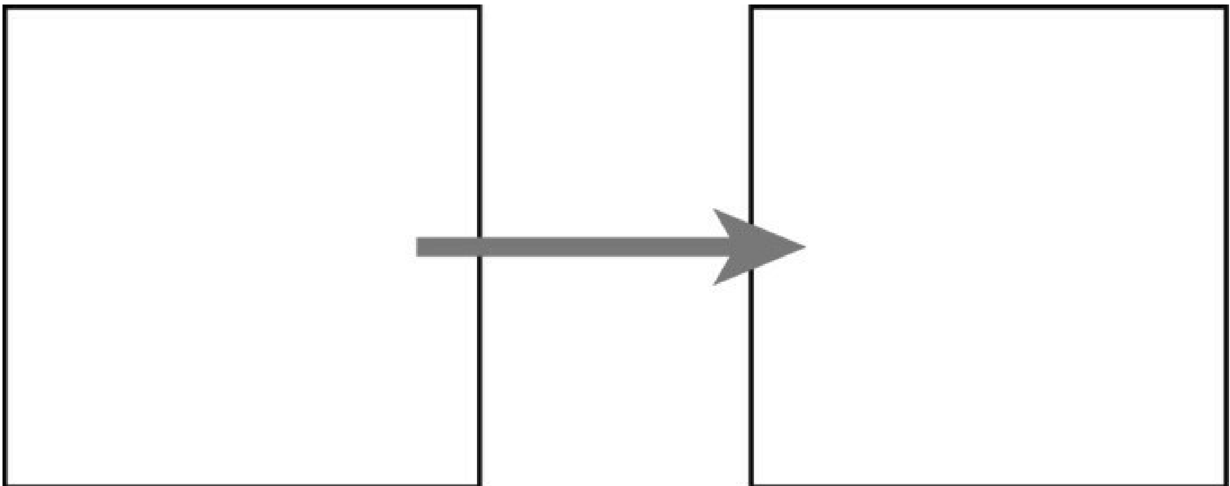


图2.6 平移示例

当选择Translate工具（热键：W）时，将注意到所选的任何对象在Scene视图中将稍微有所改变。更确切地讲，将会看到3个箭头出现，它们沿着3根轴从对象的中心指向外面。这些是平移小工具，它们有助于在屏幕上四处移动对象。单击并按住其中任何一个轴箭头将导致它们变

成黄色。然后，如果移动鼠标，对象将沿着那根轴移动。图 2.7 显示了平移小工具的样子。注意，这个些工具只会出现在Scene视图中，如果位于Game视图中，将不会看到它们。

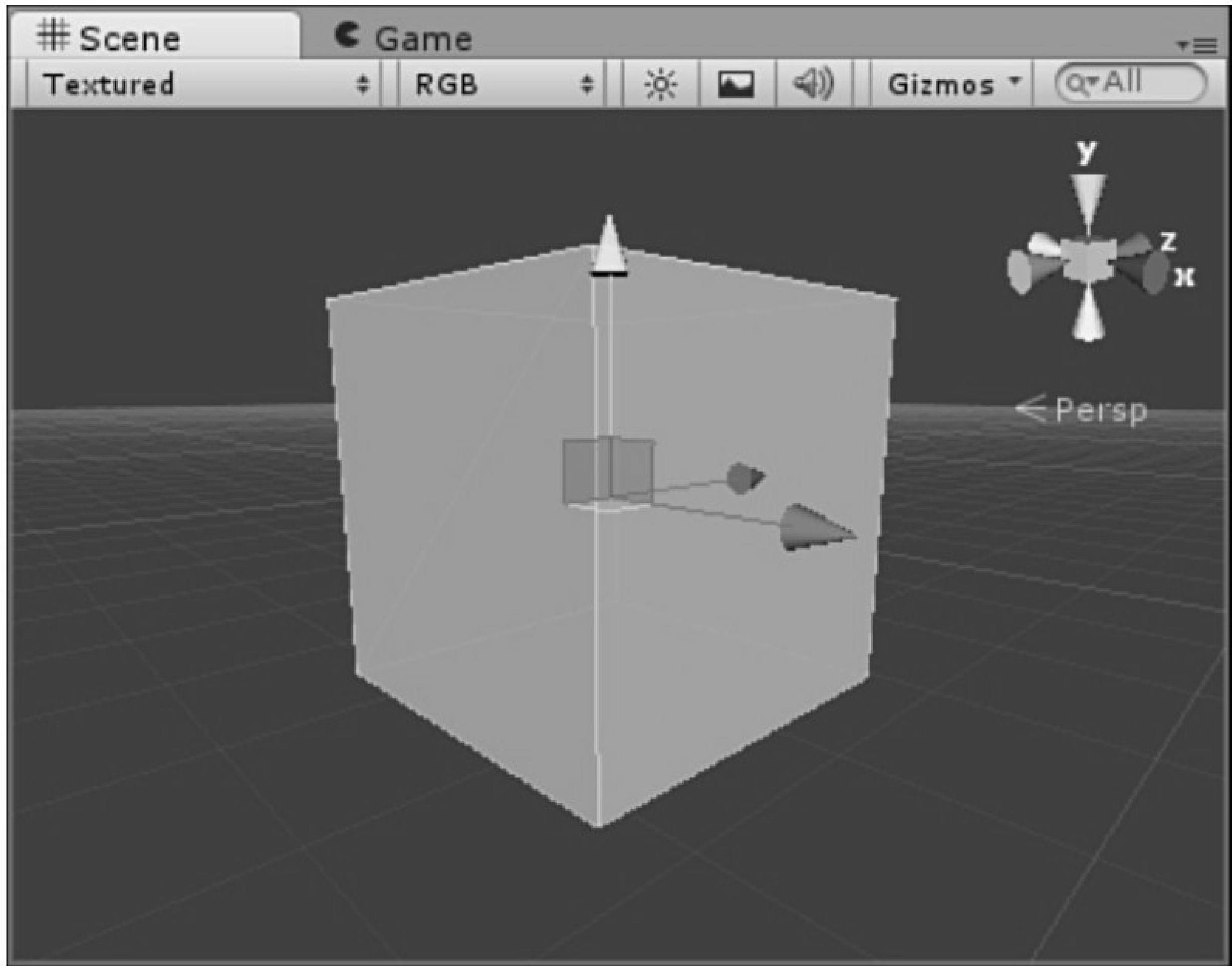


图2.7 平移小工具

提示：

变换组件和变换工具

Unity 提供了两种方式来管理对象的变换。知道何时使用每种方式很重要。你将注意到：当利用变换工具在Scene视图中改变对象的变换时，变换数据也会在Inspector视图中改变。使用Inspector视图通常更容易对对象的变换执行较大的改变，因为可以只更改它们所需要的值。不过，变换工具对于快速、较小的改变更有用。学习结合使用这两种方式

将可以极大地改进工作流程。

2.3.2 旋转

旋转一个对象不会在空间中移动它。相反，它会改变对象与那个空间的关系。更简单地讲，旋转使你能够重新定义x轴、y轴和z轴对于特定的对象指向的是哪个方向。图2.8显示了一个围绕z轴旋转的正方形。

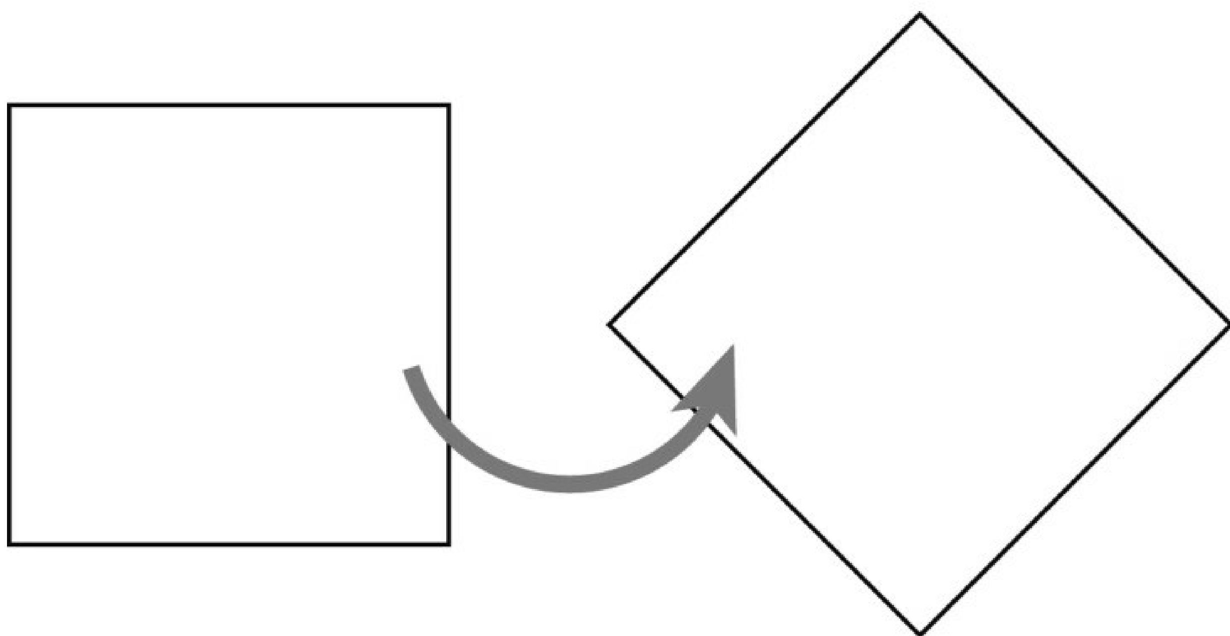


图2.8 围绕z轴旋转

提示：

确定旋转的轴

如果不确定需要围绕哪根轴旋转对象以获得想要的效果，就可以使用一种简单的方法。一次围绕一根轴旋转，假装对象就像是通过与那根轴平行的针固定在某个位置一样。对象只能围绕固定它的针旋转。现在，确定哪个针允许对象以你想要的方式旋转，它就是旋转对象所需要的轴。

就像Translate工具一样，选择Rotate工具（热键：E）将导致旋转小工具出现在对象周围。这些小工具是圆形，代表对象围绕轴的旋转路

径。单击并拖动其中任何圆形都将把它们变成黄色，并围绕那根轴旋转对象。图2.9显示了旋转小工具的样子。

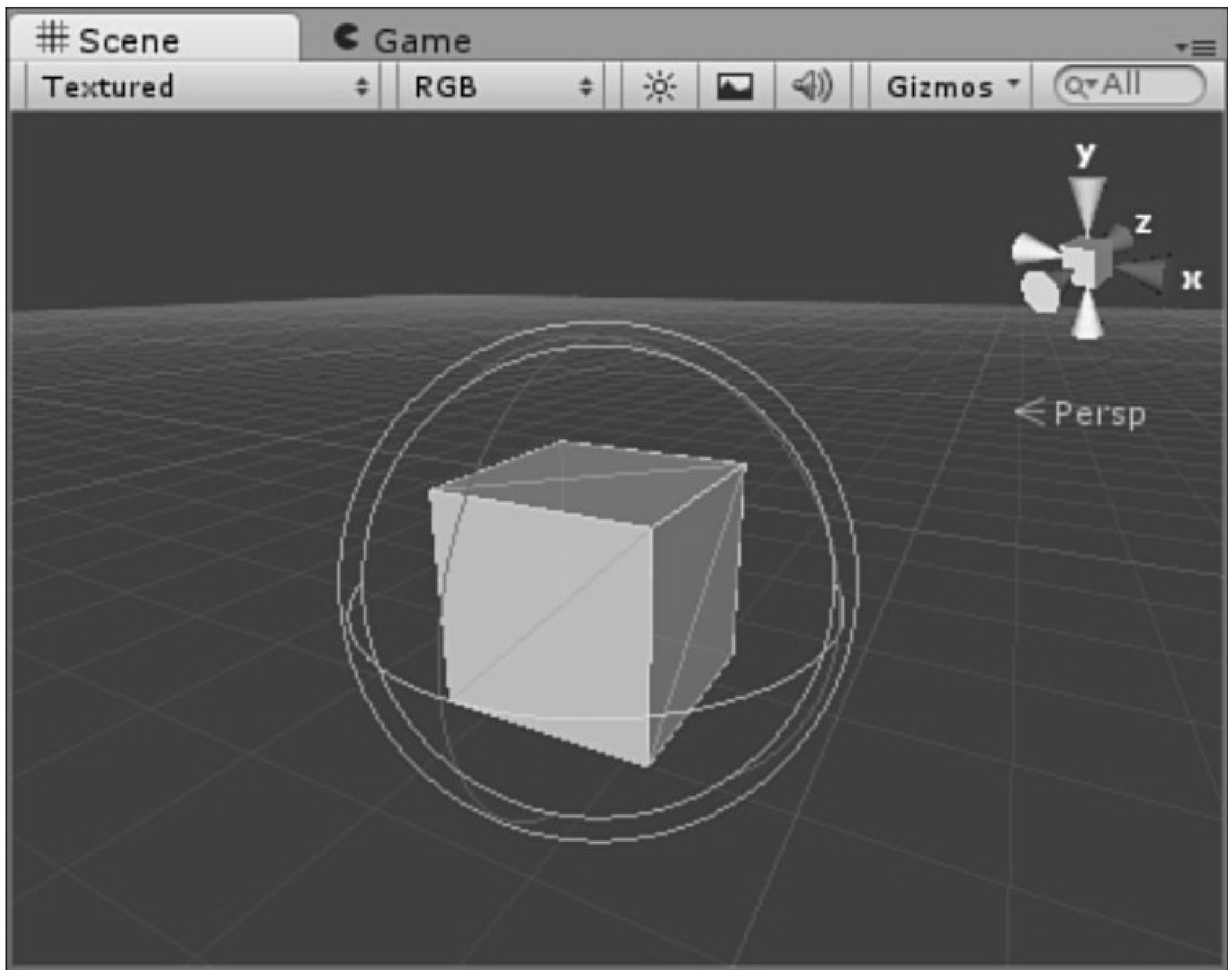


图2.9 旋转小工具

2.3.3 缩放

缩放将导致对象在3D空间内扩展或收缩。这种变换在使用时确实比较直观和简单。在任何轴上缩放一个对象都将导致它的大小在那根轴上改变。如图2.10所示，在x轴和y轴上缩小一个正方形。图2.11显示了当选择Scaling工具（热键：R）时缩放小工具的样子。

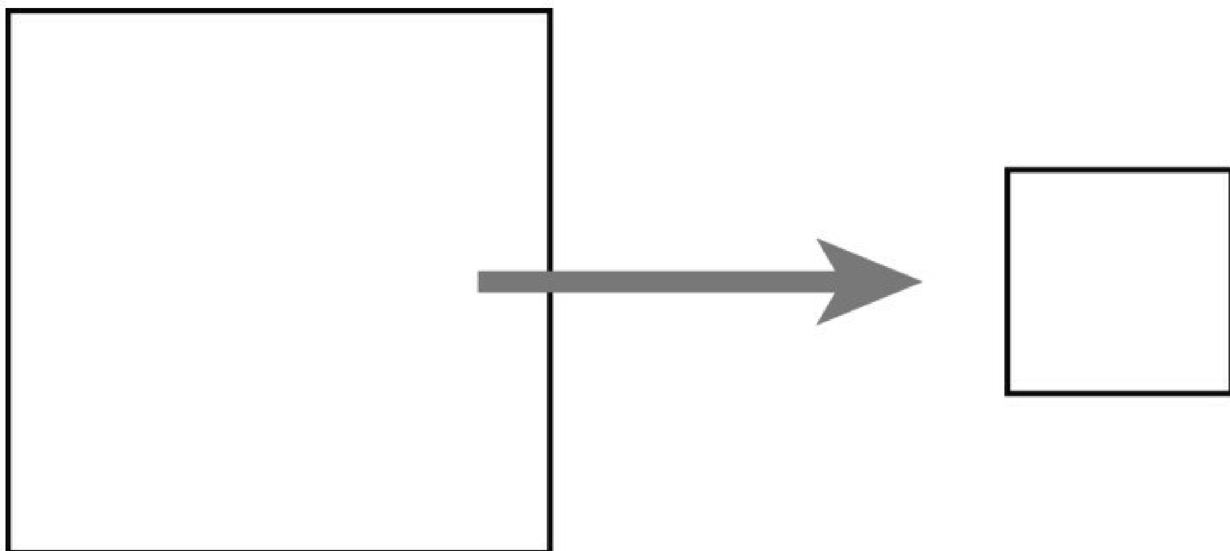


图2.10 在x轴和y轴上缩放

图2.11 缩放小工具

2.3.4 变换的风险

如前所述，变换使用局部坐标系统。因此，所做的改变可能潜在地影响将来的变换，如图2.12所示。注意当以相反的顺序应用两种相同的变换时，它们将具有非常不同的效果。

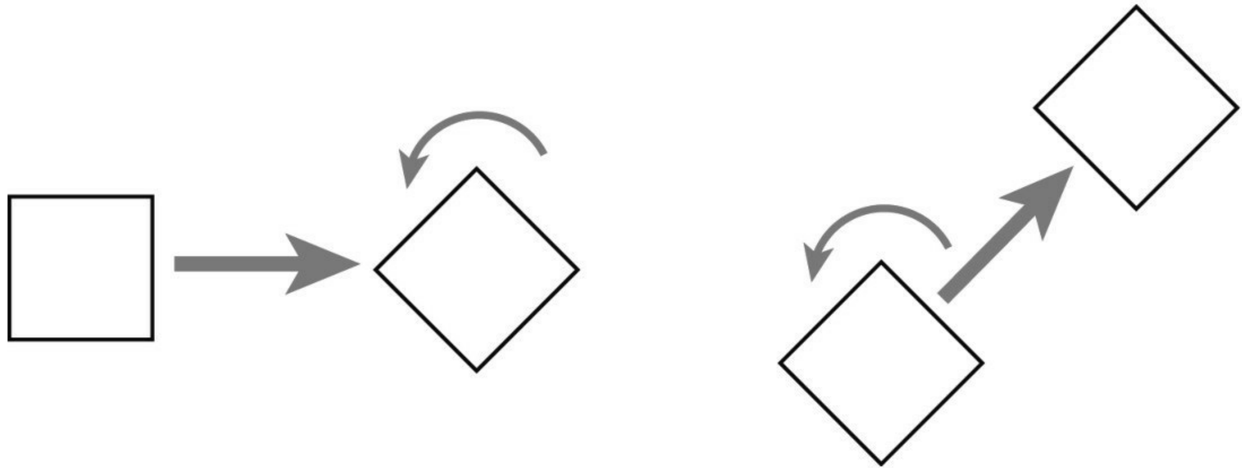


图2.12 变换顺序的效果

可以看到，不注意变换顺序将可能产生意料之外的后果。幸运的是，变换具有可以计划的一致效果。

平移：平移是一种惰性相当强的变换，这意味着在它之后应用的任何改变一般不会受其影响。

旋转：旋转将改变局部坐标系统的轴的方向。在旋转之后应用的平移将导致对象沿着新的轴移动。例如，如果要围绕z轴把对象旋转180°，然后在正y方向上移动，对象看上去好像是在向下（而不是向上）移动。

缩放：缩放实际上会改变局部坐标网格（grid）的大小。实质上讲，当把对象放大时，实际上是在把局部坐标系统放大，这导致对象似乎扩大了。这种改变具有乘法效应。例如，如果把一个对象缩放到1（它的自然、默认的大小），然后沿着x轴平移5个单位，对象看上去好像右移了5个单位。不过，如果把同一个对象缩放到2，然后在x轴上

平移5个单位，这将导致对象看上去好像右移了10个单位。这是由于局部坐标系统的大小现在加倍了，而5×2=10。相反，如果把对象缩放到0.5然后移动，它好像只移动了2.5个单位（0.5×5=2.5）。

一旦理解了这些规则，就很容易确定一组变换将如何改变对象。

2.3.5 变换和嵌套的对象

在第1章中，你学习了如何在 Hierarchy 视图中嵌套游戏对象（把一个对象拖到另一个对象上），并且这样做将稍微改变变换的工作方式。回忆可知：当把一个对象嵌套在另一个对象内时，顶级对象是父对象，而另一个对象是子对象。应用于父对象的变换将像正常的那样工作。对象可以移动、缩放和旋转。特别的是子对象的行为方式。一旦嵌套，子对象的变换就是相对于父对象进行的。因此，子对象的位置不是基于它与原点的距离，而是它与父对象的距离。如果旋转了父对象，子对象将随之移动。不过，如果查看子对象的旋转，它根本不会记录自己旋转过了。对于缩放也是如此。如果缩放父对象，子对象也会改变大小。子对象的缩放将保持不变。你可能感到迷惑的是为什么会这样。记住，当应用变换时，不是把它应用于对象，而是应用于对象的坐标系统。旋转的不是对象，而是它的坐标系统，其效果就相当于对象旋转了。当子对象的坐标系统基于父对象的局部坐标系统时，对父对象所做的任何改变都将直接改变子对象（而无需子对象知道它）。

2.4 小结

在本章中，你学习了关于Unity中的游戏对象的知识。首先学习了2D与3D之间的区别。接着，你查看了坐标系统，以及它如何从数学上分解“世界”的概念。然后，你开始处理游戏对象，包括一些内置的游戏对象。最后，你学习了有关变换的知识以及3种变换方式。你试验了一些变换，认识了一些风险，以及它们如何影响嵌套的对象。

2.5 问与答

问：同时学习2D和3D概念重要吗？

答：是的。甚至完全3D的游戏在技术层面仍然会利用一些2D概念。

问：我应该立即学习使用所有内置的游戏对象吗？

答：不必如此。有许多游戏对象，尝试立即全部学习它们将可能令人崩溃。花一些时间学习这里介绍的对象即可。

问：熟悉变换的最佳方式是什么？

答：练习。坚持不断地使用它们；最终，它们将变得相当自然。

2.6 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

2.6.1 问题

1．2D和3D中的“D”代表什么？

2．变换有多少种？

3．判断题：Unity没有内置的对象，必须创建你自己的对象。

4．如果你希望“把对象的一边放在”距离它的当前位置的右边5个单位，你是将旋转对象然后平移它，还是将平移对象然后旋转它？

2.6.2 答案

1．维度。

2．3种。

3．错误。Unity为你提供了许多内置的对象。

4．平移然后旋转。

2.6.3 练习

花点时间在父对象/子对象场景中试验变换的工作方式，将可以更好地理解坐标系统将如何精确地改变对象的定位方式。

1．创建一个新的场景或项目。

2．在项目中添加一个立方体，并把它放在(0, 2, −5)处。记住坐标的简写表示法。立方体应该具有以下坐标值：x=0、y=2、z=−5。在Inspector 视图中的变换组件中可以轻松地设置这些值。

3．在场景中添加一个球体，注意球体的x、y和z值。

4．在Hierarchy视图中把球体拖到立方体上，从而把球体嵌套在立方体下。注意位置值是如何改变的。球体现在是相对于立方体定位的。

5．把球体放在(0，1，0)处。注意它不会出现在原点的正上方，而是位于立方体的正上方。

6．现在试验多种变换。一定要在立方体以及球体上试验它们，查看它们对于父对象与子对象的影响有何不同。

第3章 模型、材质和纹理

在本章中你将学到：

模型的基本原理；

怎样导入自定义和预制的模型；

怎样使用材质和着色器（shader）。

在本章中，你将学习关于模型的知识，以及如何在 Unity 中使用它们。你首先将了解网格和3D对象的基本原理。接着，你将学习如何导入自己的模型，或者使用从Asset Store 获得的模型。在本章末尾，将探讨Unity的材质和着色器功能。

3.1 模型的基础知识

如果没有图形组件，那么视频游戏的视频（video）效果将不是非常强。在2D游戏中，图形由称为精灵（sprite）的平面图像组成。你只需改变这些精灵的x和y位置并按顺序翻转其中几个精灵，就可以欺骗观众的眼睛，使他们相信自己看到的是真正的运动和动画。不过，在3D游戏中，事实不是如此简单。在具有第三根轴的世界里，对象需要具有体积才能欺骗眼睛。由于游戏使用大量的对象，快速处理事情的需要非常重要。进入网格中，网格在其最简单的层面上讲是一系列互连的三角形。这些三角形彼此相连地构建于条带中，构成了基本的到非常复杂的对象。这些条带提供了模型的3D定义，可以非常快地进行处理。不过，不要担心，Unity 会为你处理所有这一切，因此你不必自己管理它。在本章后面，你将看到三解形怎样组成了 Unity 的 Scene 视图中的多种形状。

注意：

为什么是三角形？

你可能会问自己为什么 3D 对象完全由三角形组成。答案很简单。计算机把图形作为一系列的点（或者称为顶点，vertex）来处理。对象所具有的顶点越少，绘制它的速度就越快。三角形具有两个性质，使得它们成为人们衷爱的图形。第一个性质是无论何时具有单个三角形，都只需要另一个顶点，即可创建另一个三角形。要创建一个三角形，需要3个顶点，两个三角形只需要4个点，3个三角形只需要5个顶点。这使得它们非常高效。另一个性质是：通过使用这种创建三角形条带的实践，可以对任何3D对象建模。其他任何形状都不能提供这种程度的灵活性和性能。

注意：

模型还是网格？

模型（model）和网格（mesh）这两个术语非常相似，通常可以互换地使用它们。不过，它们之间也有区别。网格包含定义对象的 3D 形状的顶点信息。在谈论模型的形状或形式时，实际上指的是网格。因此，模型是包含网格的对象。模型具有一个定义其维度的网格，但它也可以包含动画、纹理、材质、着色器及其他网格。一个良好的一般规则是：如果正在处理的项目包含除顶点信息以外的其他任何内容，它就是模型，否则它就是网格。

3.1.1 内置的3D对象

Unity 带有几种基本的内置网格（或图元）可供你使用。它们倾向于是提供简单效用的简单形状，或者可以组合创建更复杂的对象。图 3.1 显示了可用的内置网格（你在前面的章节中处理过立方体和球体）。

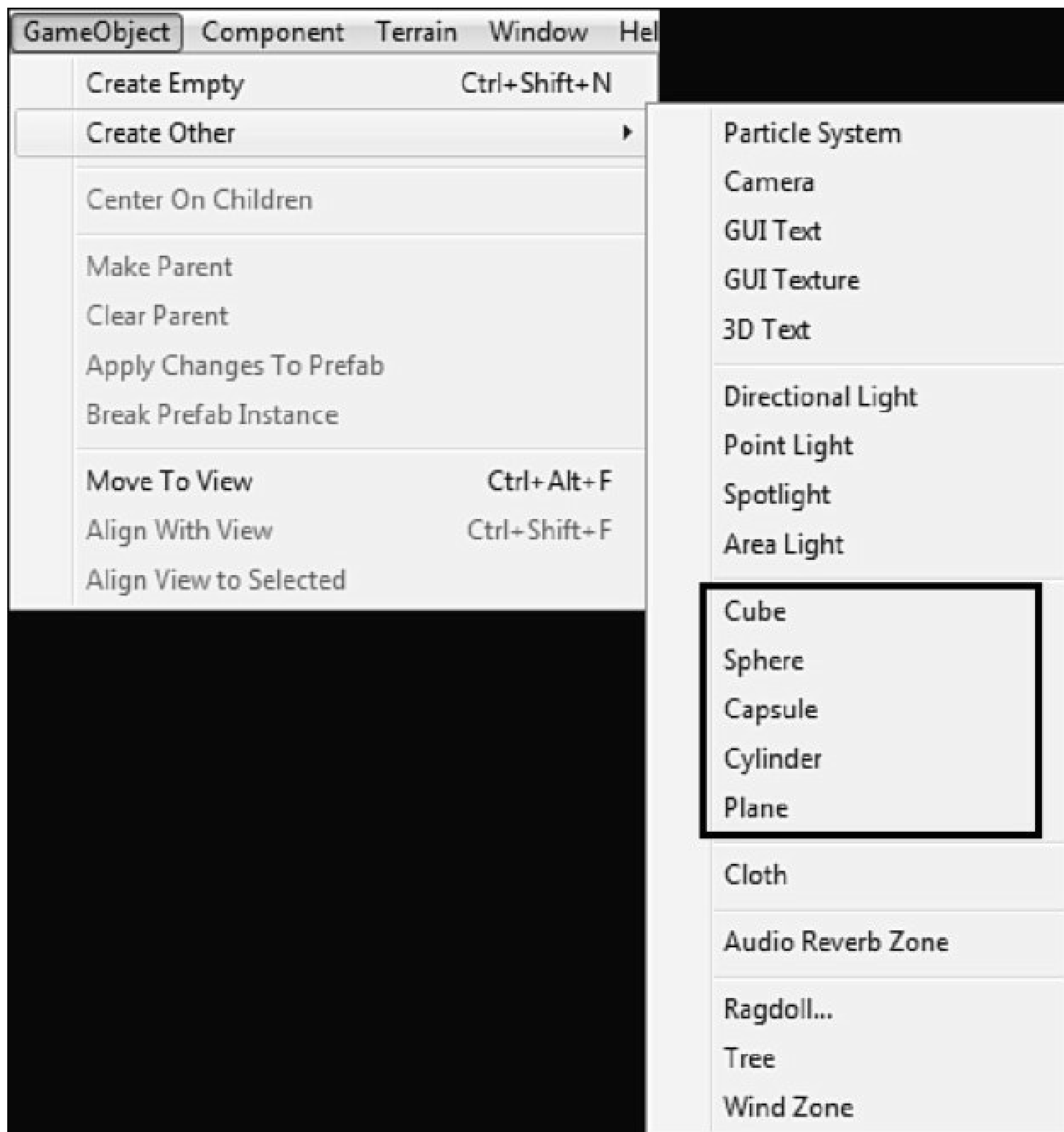


图3.1 Unity中的内置网格

提示：

利用简单的网格建模

你在游戏中需要一个复杂的对象，但却找不到要使用的适当类型的模型吗？在 Unity 中嵌套对象使你能够轻松地使用内置的网格创建简单的模型。只需把网格放在彼此相邻的位置，使它们构成你想要的粗略外

观即可。然后把所有的对象都嵌套在一个中心对象之下。这样，当移动父对象时，所有的子对象也会移动。这可能不是为游戏创建模型的最优美的方式，但它可以应一时之需。

3.1.2 导入模型

具有内置的模型很不错，但是大多数时间，游戏将需要更复杂一点的艺术资源。令人感到高兴的是，Unity使得可以相当容易地把你自己的3D模型引入到项目中。只需把包含3D模型的文件放入Assets文件夹中，就足以把它引入到项目中。从此，可以把它拖入场景或层次结构中，围绕它构建游戏对象。Unity天生支持.fbx、.dae、.3ds、.dxf和.obj文件，这使你能够使用几乎所有的3D建模工具。

导入你自己的3D模型

让我们执行把自定义的3D模型引入Unity项目中所需的步骤。

（1）创建一个新的Unity项目或场景。

（2）在Project视图中，在Assets文件夹下创建一个名为Models的新文件夹（右键单击Assets 文件夹，并选择Create > Folder命令）。

（3）在本书配套文件的Hour 3 文件夹中定位为你提供的Torus.fbx文件。

（4）并排打开操作系统的文件浏览器和Unity编辑器，在文件浏览器中单击Torus.fbx文件，并把它拖到你在第（2）步中创建的Models文件夹中。在Unity中，单击Models文件夹，查看新的Torus文件。如果操作正确，Project视图将类似于如图3.2所示的样子。注意图中为你添加了Materials文件夹，在后面将学习关于它的更多知识。

（5）在Models文件夹中单击Torus资源，并查看Inspector视图。把比例因子的值从0.01改为1，并单击Apply按钮。

（6）从Models文件夹中把Torus资源拖到Scene视图上。注意如何

把Torus游戏对象添加到包含网格过滤器和渲染网格的场景中，它们允许把 Torus 绘制到屏幕上。如果单击Torus对象，将会看到它是怎样由许多相连的三角形组成的。

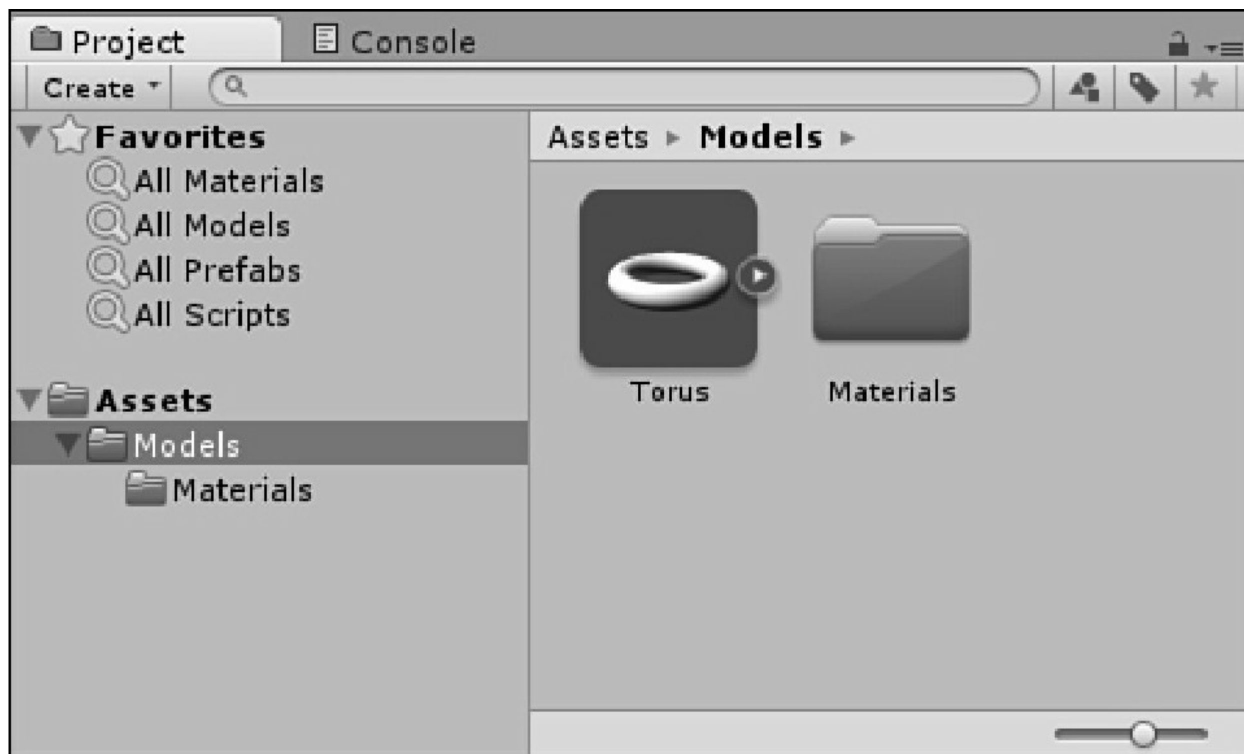


图3.2 添加了Torus模型之后的 Project视图

警告：

网格的默认缩放

用于网格的大多数Inspector视图选项都是高级选项，不会立即加以介绍。你感兴趣的属性是比例因子。默认情况下，Unity会导入缩小的网格。通过把比例因子的值从0.01改为1，告诉Unity允许模型以创建时的相同大小进入场景中。

3.1.3 模型和Asset Store

你不必成为一位建模专家，也能利用 Unity 创建游戏。Asset Store提供了一种简单有效的方式来查找预制的模型，并把它们导入到项目

中。一般来，Asset Store 上的模型分免费或付费两种，并且它们要么是独立的，要么存在于相似模型的集合中。一些模型带有它们自己的纹理，其中一些只是网格数据。

从Asset Store 下载模型

让我们学习如何从 Unity 的 Asset Store 查找和下载模型。我们将获得一个名为 Robot Kyle的模型，并把它导入到我们的场景中。

（1）创建一个新场景（单击File > New Scene 命令）。在Project 视图中，在搜索栏中输入t:Model，如图3.3所示。

（2）在搜索筛选区域中，单击Asset Store: 999+/999+按钮，如图3.3 所示。如果这些单词不可见，可能需要调整编辑器窗口或者Project视图窗口的大小。

（3）定位Robot Kyle 并选取它。



图3.3 用于定位模型资源的步骤

（4）在 Inspector 视图中，单击 Import Package。此时，可能提示你提供你的 Unity 账户凭证（account credential）。

（5）在Import Package 对话框打开时，保持一切都处于选中状态，并选择Import。

（6）将出现一个名为Robot Kyle 的新资源文件夹。在Assets > Robot Kyle > Model 下定位机器人模型，并把它拖到Scene视图中，如图

3.4所示。注意模型在Scene视图中将相当小，可能需要移近一些以查看它。

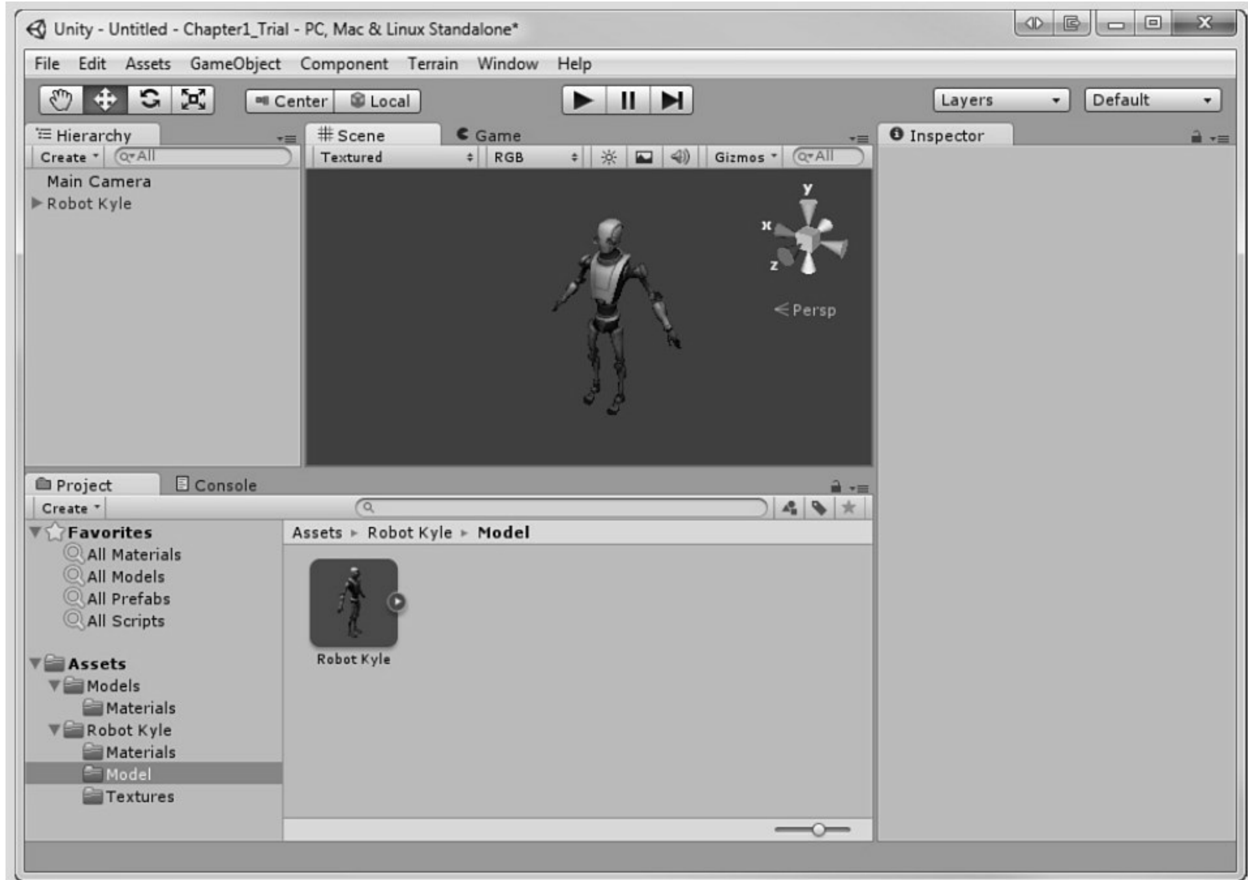


图3.4 添加了Robot Kyle的Unity项目

3.2 纹理、着色器和材质

如果不熟悉图形资源，那么在把它应用于3D模型时可能令人畏缩不前。Unity使用一种简单、特定的工作流程，它提供了许多能力，可以精确确定你希望事物看起来像什么样子的。图形资源可以分解为纹理、着色器和材质。其中每个主题都将在属于它自己的一节中单独介绍，图 3.5 显示了它们是如何密切协作的。注意，不要直接把纹理应用于模型，而是把纹理和着色器应用于材质。然后反过来又把那些材质应用于模型。这样，无需做许多工作，就能快速、干净地交换或修改模型的外观。

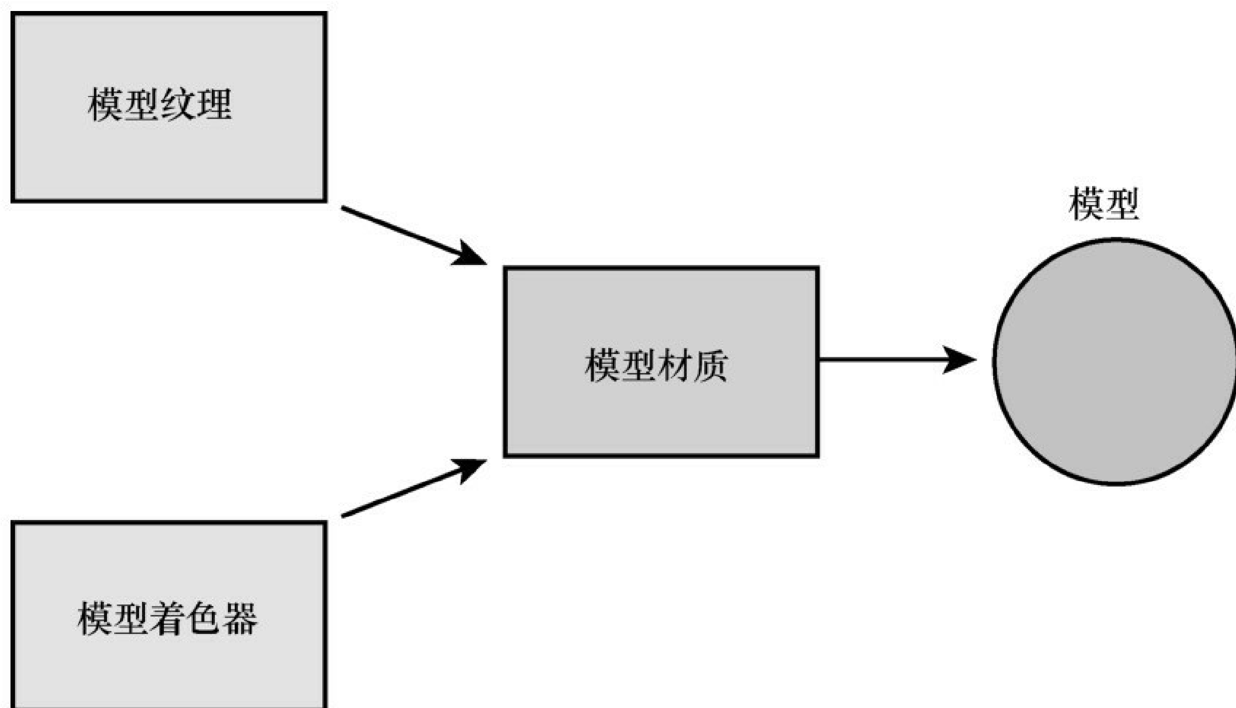


图3.5 模型资源的工作流程

3.2.1 纹理

纹理是应用于 3D 对象的平面图像。它们负责将模型变成彩色的和有趣的，而不是单调的和令人厌烦的。试图将2D图像应用于3D模型的行为可能有些奇怪，但是一旦你熟悉了它，它就会是一个相当直观的过程。我们来看一个汤罐头。如果你要是取下罐头的标签，将会看到它是一张平纸。这个标签就像是纹理。在打印标签之后，把它包装在罐头周围，提供更令人愉悦的外观。

就像所有其他的资源一样，很容易在 Unity 项目中添加纹理。首先为纹理创建一个文件夹，Textures 是一个不错的文件夹名称。然后把你想要添加到项目中的任何纹理拖到你刚刚创建的Textures文件夹中。这就行了！

注意：

那是打开包装！

想象一下纹理怎样包装在罐头周围，但是对于更复杂的对象则该如何？在创建一个错综复杂的模型时，生成所谓的打开包装（unwrap）就比较常见。打开包装有点像贴图，准确显示了平面纹理将如何包装在模型周围。如果查看本章前面的 Robot Kyle > Textures 文件夹，就会注意到Robot_Color纹理。它看上去比较奇怪，但这是用于模型的打开包装的纹理。打开包装、模型和纹理的生成自身是一种艺术形式，将不会在本书中介绍。初步了解它是如何工作的在这个层面应该就足够了。

警告：

怪异的纹理

在本章后面，将对模型应用一些纹理。你可能注意到纹理有点弯曲或者在错误的方向上翻转，只需知道这不是一个错误。在获取基本的矩形2D纹理并把它应用于模型时，就会发生这个问题。模型不知道哪种方式是正确的，因此它将尽其所能应用纹理。如果想避免这个问题，可以使用专门为你正在使用的模型设计（打开包装）的纹理。

3.2.2 着色器

如果模型的纹理确定了在它表面绘制什么内容，那么着色器就确定了如何绘制它。看待它的另一种方式是：材质包含属性和纹理，着色器则规定了材质可以具有什么属性和纹理。这在此刻似乎可能是无意义的，但是以后当我们创建材质时，你将开始理解它们是如何工作的。在本章后面将介绍关于着色器的大量信息，因为不能在没有材质的情况下创建着色器。事实上，在学习的关于材质的大量信息实际上是关于材质的着色器的信息。

提示：

思考练习

如果在理解着色器的工作方式时比较费力，可以考虑下面这种场景：假设你有一块木头。木头的物理性是它的网格，颜色、质地和可见的元素是它的纹理。不过，它看上去有所不同。它稍微暗一点，并且有光泽。这个示例中的水就是着色器，着色器具有某种特性，使之看上去有点不同，而不会实际地改变它。

3.2.3 材质

如前所述，材质差不多就是可以应用于模型的着色器和纹理的容器。自定义材质的大部分工作就是为它选择哪种着色器，尽管所有的着色器都具有一些共同的功能。

要创建一种新材质，首先要创建一个Materials文件夹。然后右键单击该文件夹，并选择Create > Material 命令。给材质提供某个描述性的名称，这就行了。图3.6显示了两种选择了不同着色器的材质。注意其中每种材质都具有基本纹理、主色、拼接（tilling）和偏移量（offset），以及材质的预览（现在是空白的，因为没有纹理）。不

过，Shiny 材质使用一种高光着色器（specular shader），并且带有用于高光颜色（specular color）和光泽的属性。所有这些属性都将在本章后面介绍。

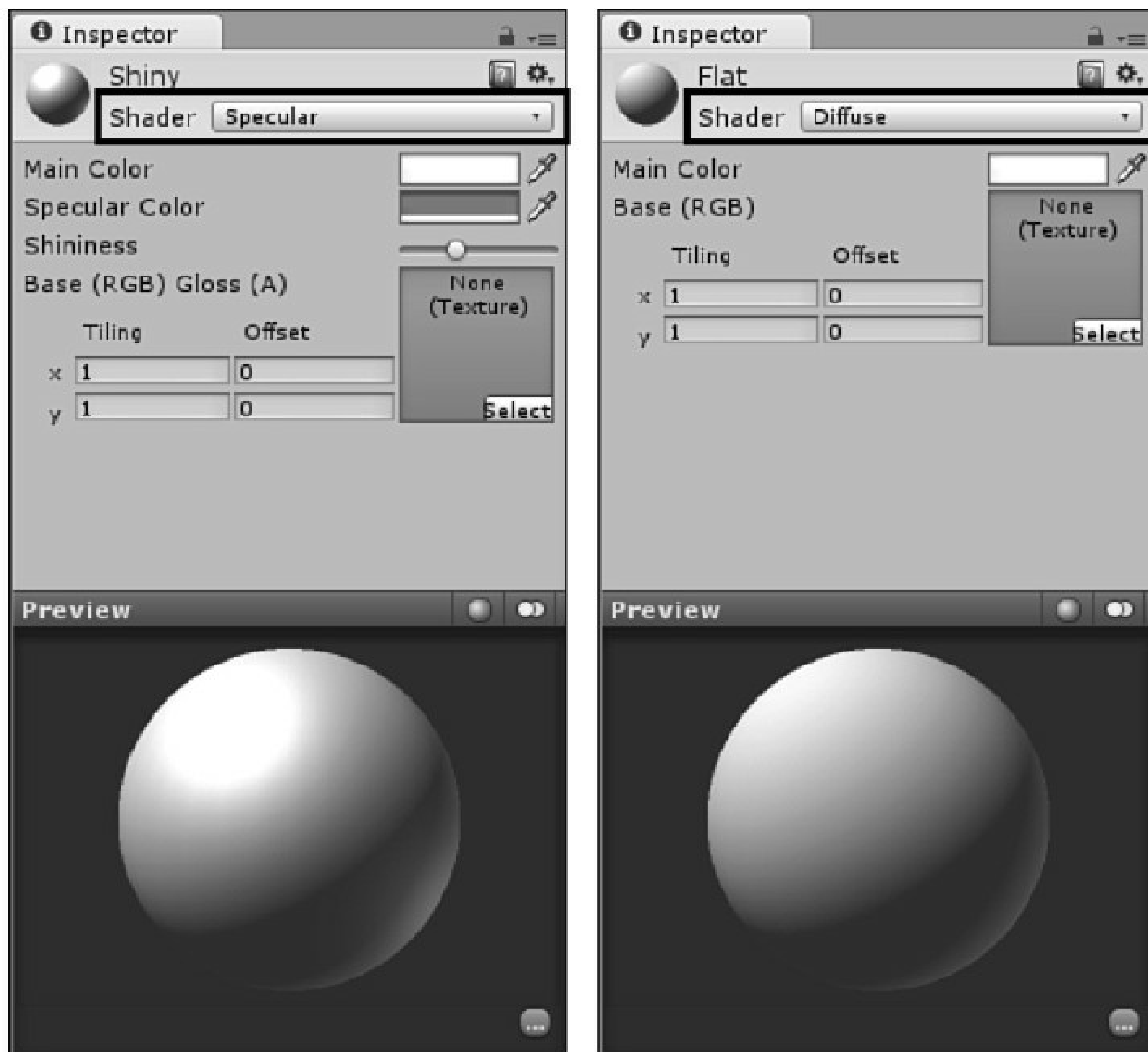


图3.6 两种具有不同着色器的材质

3.2.4 再论着色器

既然你已经理解了纹理、模型和着色器，现在就应该查看怎样把它们结合在一起。Unity具有许多内置的着色器，但是本书只涉及 Normal

系列中的少数几种着色器。这些着色器是最基本的，并且对每一个人都有用。表3.1描述了一些基本的着色器。

表3.1 基本的Normal系列的着色器

着色器	描述
漫反射着色器	漫反射是用于材质的默认着色器，并且也是最基本的。灯光均匀分布在漫反射对象的整个表面上
高光着色器	高光纹理使对象看上去像在发光。如果你想使对象看上去似乎在反射许多灯光，就要使用这种着色器
凹凸着色器	凹凸着色器一般与其他着色器一起使用（比如凹凸—漫反射或凹凸—高光）。这些着色器使用法线贴图给平面纹理提供一种 3D 或凹凸外观。它们是给模型提供许多物理细节的极佳方式，而无需复杂的建模

既然你已经熟悉了几种内置的着色器，现在就应该探讨你将使用的一些常用的着色器属性。表3.2描述了常用的着色器属性。

表3.2 常用的着色器属性

属性	描述
Main Color	Main Color 属性定义了在对象上发出的是什么颜色的环境灯光。这不会改变对象自身的颜色，它只使对象看上去有所不同。例如，具有蓝色纹理和黄色主色的对象将不会变成黄色，但是会变成绿色（因为蓝色与黄色灯光在一起看上去像绿色）。如果想要模型的颜色保持不变，可以选择白色
Specular Color	Specular Color 属性确定高光模型的“发光”部分是什么颜色。一般来讲，这将是白色的，除非打算让它看上去好像另一种颜色的灯光在照亮对象
Shininess	Shininess 属性确定高光对象是多么闪亮
Texture	Texture 块包含你想应用于模型的纹理
Normal Map	Normal Map 块包含将应用于模型的法线贴图。法线贴图可用于对模型应用凹凸效果。在计算光照给模型提供了比它所具有的更多细节时，这是有用的
Tiling	Tiling 属性定义了纹理可以在模型上多久重复一次。它可以同时在 x 轴和 y 轴上重复
Offset	Offset 属性定义了在对象与纹理的边缘之间是否存在间隙

这看上去似乎有许多信息要吸收，但是一旦你对纹理、着色器和材质的少量基础知识变得更熟悉，将会发现这是一个顺畅的过程。

对模型应用纹理、着色器和材质

让我们把所掌握的关于纹理、着色器和材质的所有知识结合起来，创建一种看上去相当不错的砖墙效果。

（1）开始一个新项目或场景。注意：创建新项目将会关闭并重新打开编辑器。

（2）创建一个Textures和一个Materials文件夹。

（3）在本书配套文件中定位 Brick_Texture.png 文件，并把它拖到在第（2）步中创建的Textures文件夹中。

（4）向场景中添加一个立方体，把它定位在(0, 1, −5)处，并给它提供(5, 2, 1)的比例。参见图3.7来了解立方体的属性。

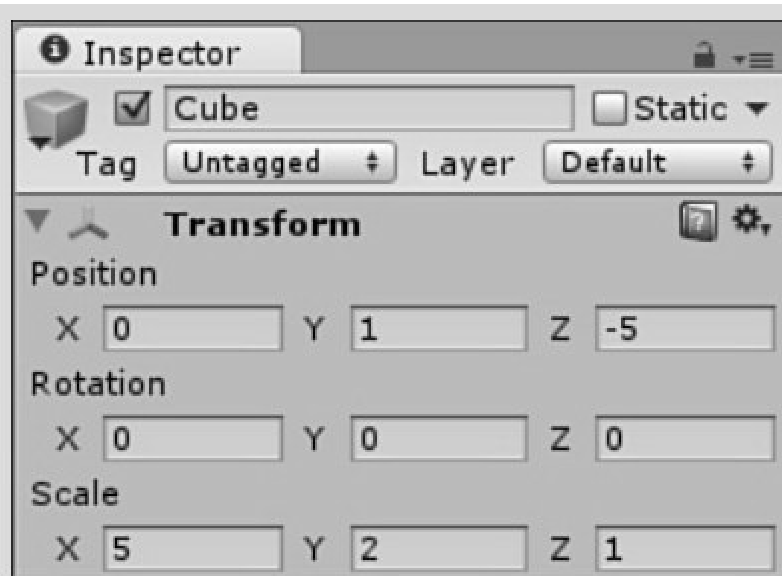


图3.7 立方体的属性

（5）创建一种新材质（右键单击Materials 文件夹，并选择Create > Material 命令），并把它命名为BrickWall。

（6）保持着色器为 Diffuse，并在纹理块中单击 Select 命令。从弹出式窗口中选择Brick_Texture命令。

（7）把砖墙材质从Project视图中拖到Scene视图中的立方体上。

（8）注意纹理在墙面上是怎样被拉伸得更大一点的。在选择了材质的情况下，把x拼贴的值改为3。注意墙面看上去要好得多。

（9）向场景中添加一种定向灯光（单击GameObject > Create Other > Directional Light命令）。把它定位在(0, 10, –10)处，并把旋转角度设置为(30, 0, 0)。在后面一章中将更多地讨论光照。这里只是为了使砖墙“凸出来”。

(10）场景中现在将具有纹理化的砖墙。图3.8所示为最终的作品。

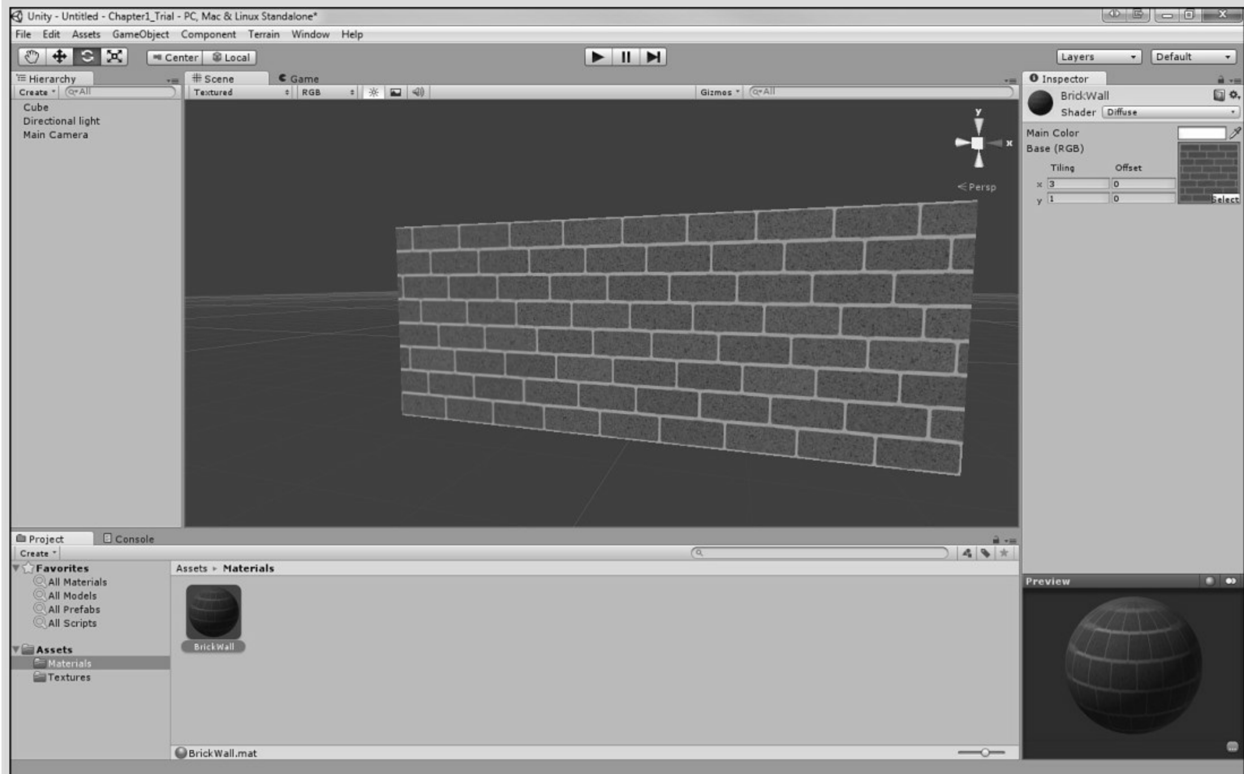


图3.8 这个Try It Yourself的最终作品

3.3 小结

在本章中，你认识了 Unity 中的模型。首先学习了如何利用称为网格顶点的集合构建模型。然后，你发现了如何使用内置的模型，导入你自己的模型，以及从Asset Store 下载模型。你接着学习了 Unity 中的模型艺术工作流程。你试验了纹理、着色器和材质，最后通过创建一块纹理化的砖墙结束本章的学习。

3.4 问与答

问：如果我不是艺术家，那我仍然能够创建游戏吗？

答：绝对可以。使用免费的在线资源和Unity Asset Store，可以找到多种艺术资源放到游戏中。

问：我需要知道如何使用所有内置的着色器吗？

答：不必如此。许多着色器与实际情况有很大的关系。从本章中介绍的着色器开始，如果游戏项目需要，可以再学习更多的着色器。

问：如果 Unity Asset Store 中有付费的艺术资源，这意味着我可以出售自己的艺术资源吗？

答：是的，可以这样做。事实上，它并不仅限于艺术资源。如果你可以创建高质量的资源，当然可以在商店里出售它们。

3.5 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

3.5.1 问题

1．判断题：由于正方形的简单性，它们组成了模型中的网格。

2．Unity为3D模型支持哪些文件格式？

3．判断题：从Unity Asset Store 只能下载付费的模型。

4．解释纹理、着色器和材质之间的关系。

3.5.2 答案

1．错误，网格是由三角形组成的。

2．.fbx、.dae、.3ds、.dxf和.obj文件。

3．错误。有多种免费的模型。

4．材质包含纹理和着色器，着色器规定了可以通过材质设置的属性以及如何渲染材质。

3.5.3 练习

让我们试验着色器对模型的外观所产生的影响。你将为每种模型使用相同的网格和纹理，只有着色器将是不同的。在这个练习中创建的项目被命名为 Hour3_Exercise，可以在本书配套文件的的Hour 3文件夹中找到它。

1．创建一个新场景或新项目。

2．在项目中添加一个Materials 或Textures 文件夹。定位本书配套

文件的Hour 3 文件夹中的Brick_Normal.png和Brick_Texture.png文件，并把它们拖到Textures文件夹中。

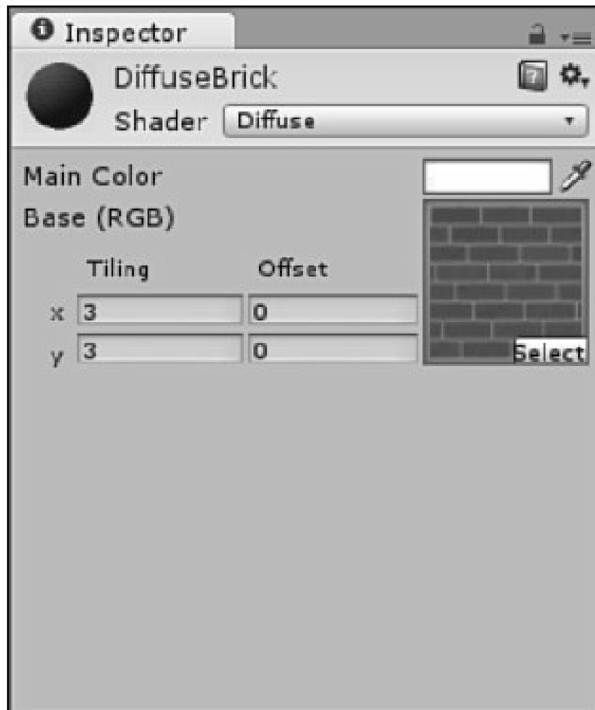
3．在Project视图中，选择Brick_Texture。在Inspector视图中，把茴香级别从1改为3，提高曲线的纹理质量。然后单击Apply按钮。

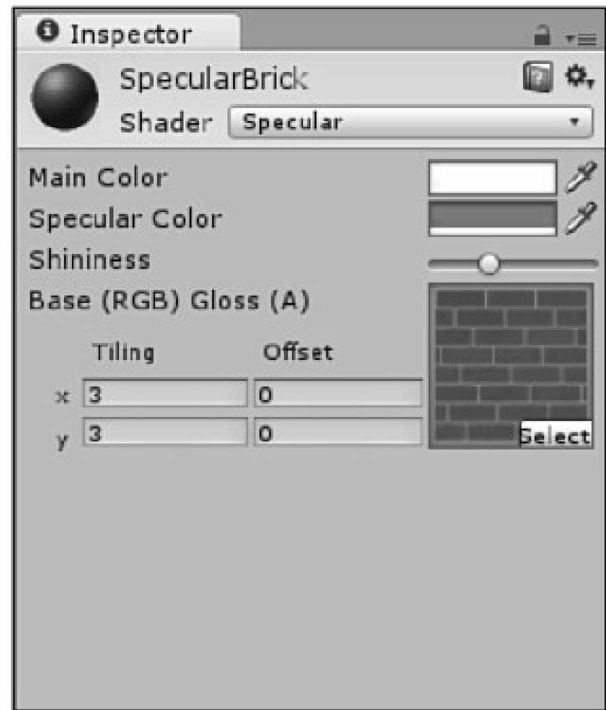
4．在Project视图中，选择Brick_Normal。在Inspector视图中，把纹理类型改为Normal Map。然后单击Apply按钮。

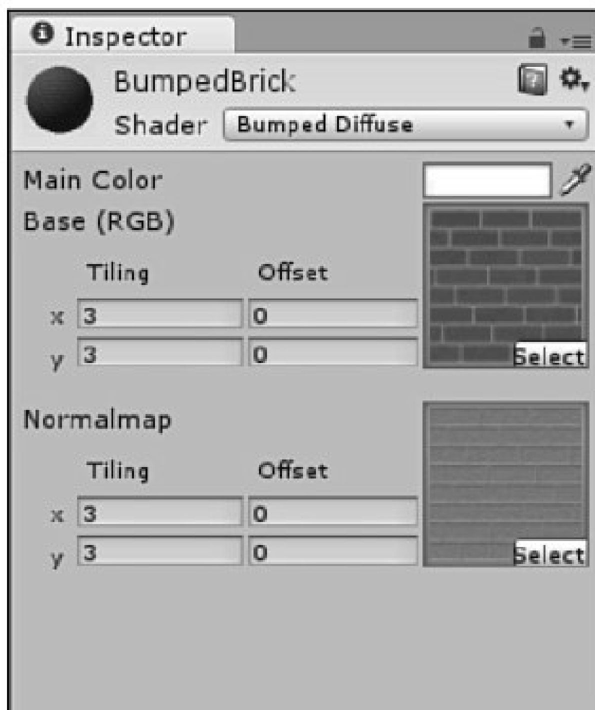
5．在项目中添加定向灯光（单击GameObject > Create Other > Directional Light 命令），把它的位置设置为(0, 10, −10)，并设置旋转角度为(30, 40, 0)。

6．在项目中添加4 个球体，并把它们都缩放为(2, 2, 2)。然后通过把它们的位置分别设置为(2, 0, −5)、(−2, 0, −5)、(−2, 2, −5)和(2, 2, −5)，把它们分散开。

7．在 Materials 文件夹中创建4种新材质，并把它们分别命名为DiffuseBrick、SpecularBrick、BumpedBrick和BumpedSpecularBrick。图3.9显示了4种材质的所有属性，继续前进并设置它们的值。







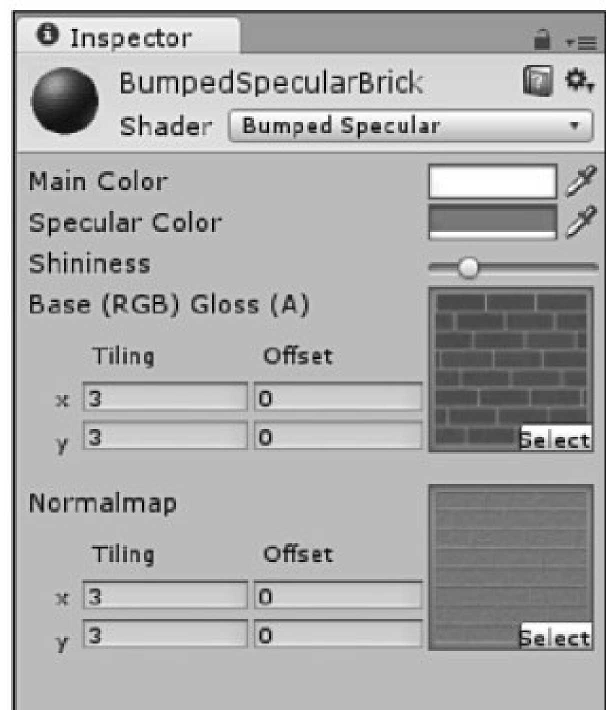


图3.9 材质的属性

8．单击每种材质并把它拖到4个球体之一上。注意球体的灯光和曲度怎样与不同的着色器交互。记住，可以在Scene视图中四处移动，以不同的角度查看球体。

第4章 地形

在本章中你将学到：

地形的基本原理；

怎样雕刻地形；

怎样利用纹理装饰地形。

在本章中，你将了解地形生成。你将学习地形是什么，怎样创建它，以及怎样雕刻它。你还将实践纹理绘画和微调。此外，还将学习为游戏创建宽广、辽阔和逼真的地形。

4.1 地形生成

所有的 3D 游戏关卡都存在于某种形式的游戏世界里。这些游戏世界可以是高度抽象或逼真的。通常，带有辽阔的“室外”关卡的游戏被称为具有一种地形。术语地形（terrain）指模拟世界的外部风景的任何陆地区域。高高的山脉、一望无际的平原或者潮湿的沼泽地都是可能的游戏地形的示例。

在 Unity 中，地形是可以雕刻成许多不同形状的平面网格。把地形视作沙箱里的沙子可能会有帮助。你可以挖掘沙子，或者使它的某些部分隆起。基本地形唯一不能做的是交叠，这意味着不能创建像洞穴或突出物这样的物体。这些项目必须单独建模。此外，就像 Unity中的其他任何对象一样，地形也具有位置、旋转和比例（尽管它们通常不会改变）。

4.1.1 在项目中添加地形

在场景中创建一种平面地形是设置一些基本参数的简单任务。要把地形添加到场景中，只需单击GameObject > Create Other > Terrain 命令即可。你将看到添加了一个名为Terrain 的对象。如果在Scene视图中导航，还可能注意到地形部分非常大。事实上，这个部分要比我们目前可能需要的大得多。因此，我们需要修改这个地形的一些属性。

要使这个地形更容易管理，需要更改地形分辨率。通过修改分辨率，可以更改地形部分的长度、宽度和最大高度。以后当你学习了高度图之后，将更明白使用分辨率（resolution）这个术语的原因。要更改地形部分的分辨率，可遵循下面这些步骤。

（1）在Hierarchy视图中选择你的地形。然后在Inspector视图中找到并单击Terrain Settings按钮，如图4.1所示。

（2）找到Resolution设置。

（3）目前，地形宽度和长度被设置为2000，把这些值都设置为50。

Inspector

X 1 Y 1 Z 1

Terrain (Script)

Terrain Settings

Base Terrain

Pixel Error	5
Base Map Dist.	1000
Cast shadows	☑
Material	None (Material)

Tree & Detail Objects

Draw	☑
Detail Distance	80
Detail Density	1
Tree Distance	2000
Billboard Start	50
Fade Length	5
Max Mesh Trees	50

Wind Settings

Speed	0.5
Size	0.5
Bending	0.5
Grass Tint	

Resolution

Terrain Width	50
Terrain Length	50
Terrain Height	600
Heightmap Resolutio	513
Detail Resolution	1024
Detail Resolution Per	8
Control Texture Reso	512
Base Texture Resolu	1024

* Please note that modifying the resolution will clear the heightmap, detail map or splatmap.

图4.1 Resolution设置

Resolution　设置中的其他选项用于修改纹理的绘制方式以及地形的表现方式。目前不用设置这些选项。在修改了宽度和高度之后，将会看到地形要小得多并且更容易管理。现在可以开始雕刻了。

警告：

地形大小

目前，你将处理的地形的长度和宽度都是50单位，这纯粹是为了在学习多种工具时容易进行管理。在真实的游戏中，地形可能会大得多，以适应你的需要。还值得指出的是：如果你具有高度图（将在下一节中介绍），则将希望地形比例（长度和宽度的比例）与高度图的比例匹配。

4.1.2 高度图雕刻

传统上讲，在8位图像中有256种灰色阴影可用，这些阴影的范围是0（黑色）～255　（白色）。知道这一点之后，就可以获取一幅黑白图像，通常称为灰度（grayscale）图像，并把它用作所谓的高度图（heightmap）。高度图是一幅灰度图像，其中包含与地形图类似的海拔信息。较深的阴影可能被认为是较低的位置，较浅的阴影则被认为是较高的位置。图4.2显示了一个高度图的示例。它看起来可能不是非常像，但是像这样的简单图像可以产生一些动态的风景。



图4.2 简单的高度图

对当前的平面地形应用高度图很容易，只需简单地从一种平面地形开始，并把高度图导入到其上，如下所示。

对地形应用高度图

（1）在本书配套文件的Hour 4 文件夹中定位terrain.raw文件，并把它放到你可以轻松找到的某个位置。

（2）在Hierarchy视图中选择你的地形，然后单击Terrain Settings 按

钮（如果你不记得它在哪里，参见图4.1）。在 Heightmap 区域中，单击Import Raw按钮。

（3）打开ImportRawHeightmap对话框。定位第（1）步中的terrain.raw，并单击Open按钮。

（4）打开Import Heightmap 对话框，如图4.3 所示。使所有的选项保持不变，并单击Import按钮。现在，你的地形看上去比较怪异。问题在于当你把地形的长度和宽度设置成更容易管理时，将高度保持为600，这对于你当前的需要来说显然太高了。

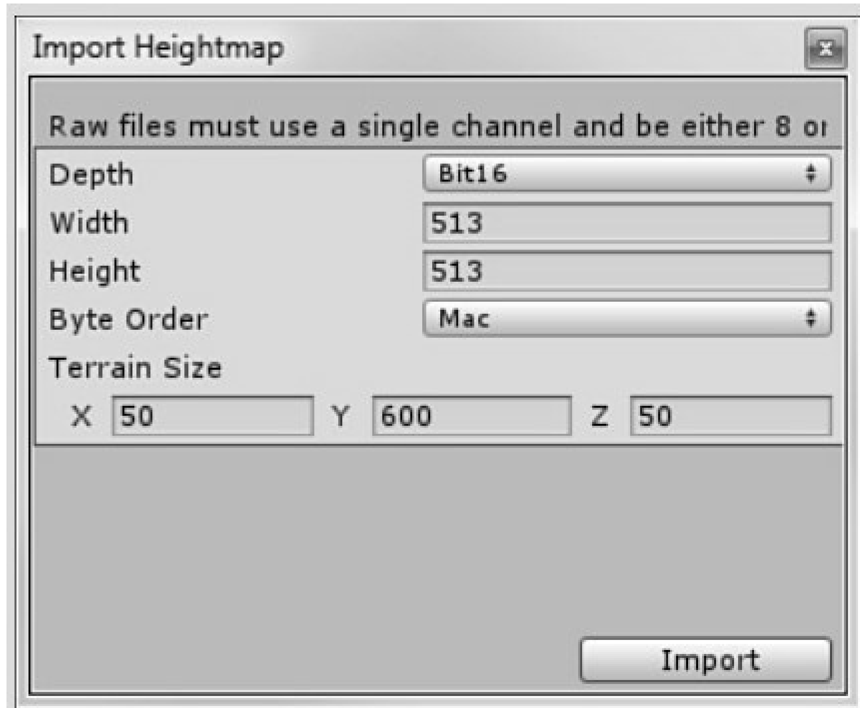


图4.3 Import Heightmap 对话框

（5）回到Inspector视图中的Terrain Settings 中的Resolution 区域，修改地形分辨率。这一次，把高度值改为60。结果应该会令人愉快得多，如图4.4所示。

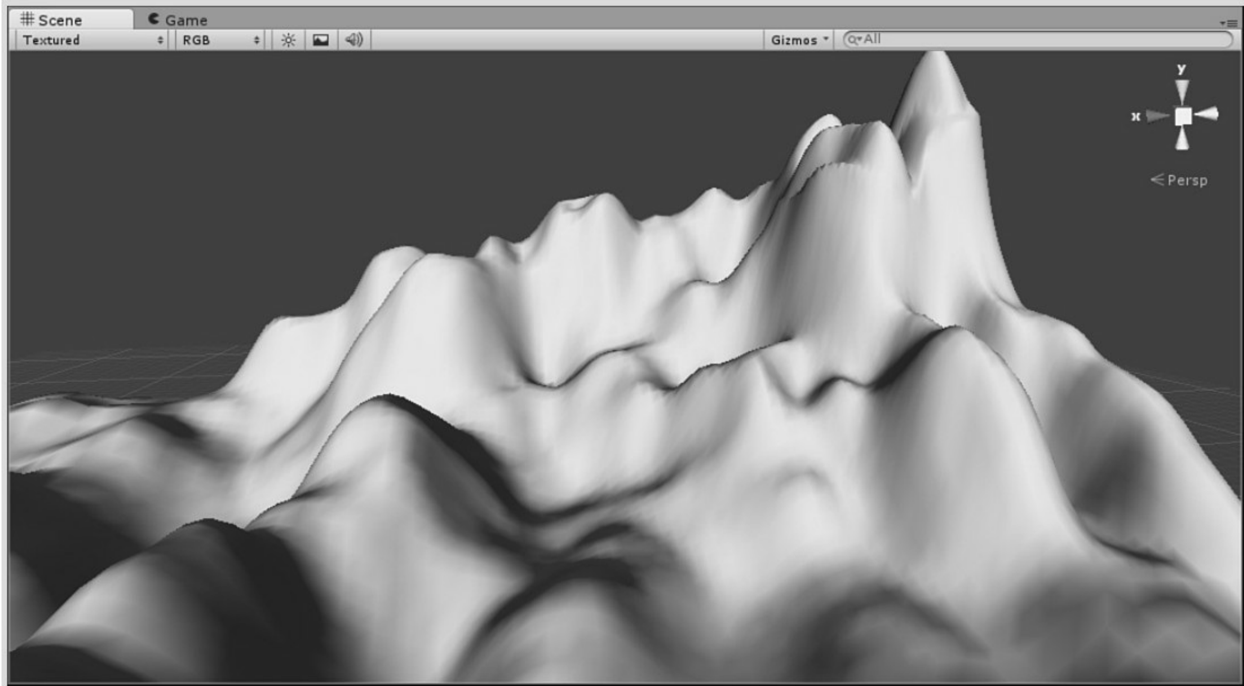


图4.4 导入高度图后的地形

提示：

计算高度

迄今为止，高度图似乎比较随机，但它实际上相当容易计算出来。所有的一切都基于255的百分比与地形的最大高度。地形的最大高度默认为600，但是很容易改变。如果应用(灰度阴影)÷255×(最大高度)这个公式，就可以轻松地计算地形上的任意位置。例如，黑色具有值0，因此任何黑色区域的高度都将是0单位（0÷255×600）。白色具有值255，因此将产生高度为600单位的区域（255÷255×600）。如果具有值为125的中等灰度，那么具有该颜色的任何区域都将产生大约294单位的高度（125÷255×600）。

注意：

高度图格式

在Unity中，高度图必须是.raw格式的灰度图像。有许多方式生成这些类型的图像，可以使用简单的图像编辑器或者甚至Unity本身。如果使用图像编辑器创建高度图，就要尽量使高度图具有与地形相同的长宽

比。否则，就会明显看出某种扭曲。如果使用Unity的雕刻工具雕刻某种地形，并且希望为它生成一幅高度图，可以转到 Inspector 视图中的Terrain Settings 中的Heightmap 区域，并单击Export Raw 按钮。

4.1.3 Unity地形雕刻工具

Unity 提供了多种工具，可让你手工雕刻地形。在 Terrain (Script)组件下的 Inspector视图中可以看到这些工具，它们都在相同的前提下工作：你使用具有给定大小的画笔和不透明度“绘制”地形。实际上，你在幕后所做的事情是绘制一幅高度图，它被转换成为3D 地形所做的修改。绘画效果是累积的，这意味着在一个区域上绘制的内容越多，那个区域上的效果将越强烈。图4.5标识了这些工具。使用这些工具，可以生成你可能想象到的几乎任何风景。

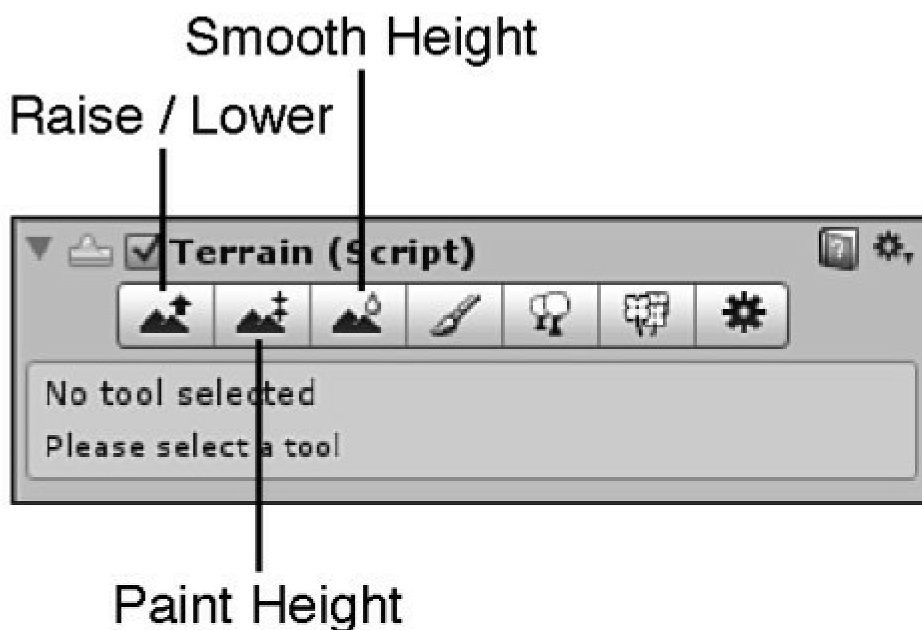


图4.5 地形雕刻工具

你将学习的第一个工具是 Raise/Lower 工具。顾名思义，无论何时你在绘画，这个工具都允许抬高或降低地形。要利用这个工具执行雕刻，可遵循下面这些步骤。

（1）选择一种画笔。画笔确定了雕刻效果的大小和形状。

（2）选择一种画笔大小和不透明度。不透明度确定了雕刻效果有多强烈。

（3）在 Scene 视图中单击地形并在其上拖动，以抬高地形。在单击并拖动时按住 Shift键，将代之以降低地形。

图4.6演示了一些良好的起始选项，它们用于雕刻大小为50×50、高度为60的给定地形。

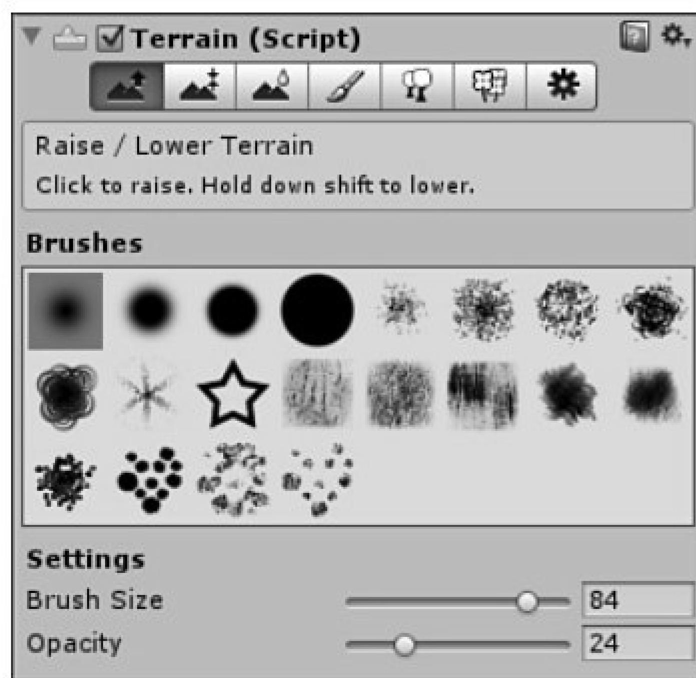


图4.6 用于雕刻的良好起始属性

下一个工具是Paint Height 工具。该工具的工作方式与Raise/Lower工具几乎完全相同，只不过它把地形绘制到指定的高度。如果指定的高度高于当前地形，那么绘制效果将抬高地形。不过，如果指定的高度低于当前地形，就会降低地形。对于在风景中创建台地或其他平坦的结构，这被证明是有用的。继续前进，并试试它！

提示：

平整地形

在任何时候，如果想把地形重置回平面形状，都可以找到 Paint

Height工具，并单击 Flatten。它的一个额外的优点是：可以把地形平整到一种除其默认的0以外的高度。如果最大高度是60，并把高度图平整到30，就能够把地形抬高30个单位，但是也可以把它降低30个单位。这使得很容易把峡谷雕刻进平面地形中。

你将使用的最后一个工具是 Smooth Height 工具。该工具不会以特别显著的方式改变地形。作为替代，它将删除在雕刻地形时出现的许多锯齿状线条。可以把该工具视作抛光机。在完成了主要的雕刻之后，你将只把它用于执行微小的调整。

雕刻地形

既然你已经学习了雕刻工具，就让我们练习使用它们。在这个练习中，你将尝试雕刻一块特定的地形。

（1）创建一个新项目或场景，并添加一种地形。把该地形的分辨率设置为 50×50，并把它的高度设置为60。

（2）单击 Paint Height 工具，把高度改为 20，并单击 Flatten，把该地形平整到高度为20。

（3）使用雕刻工具，尝试创建类似于如图4.7所示的风景。

（4）继续试验工具，尝试向地形中添加独特的特性。

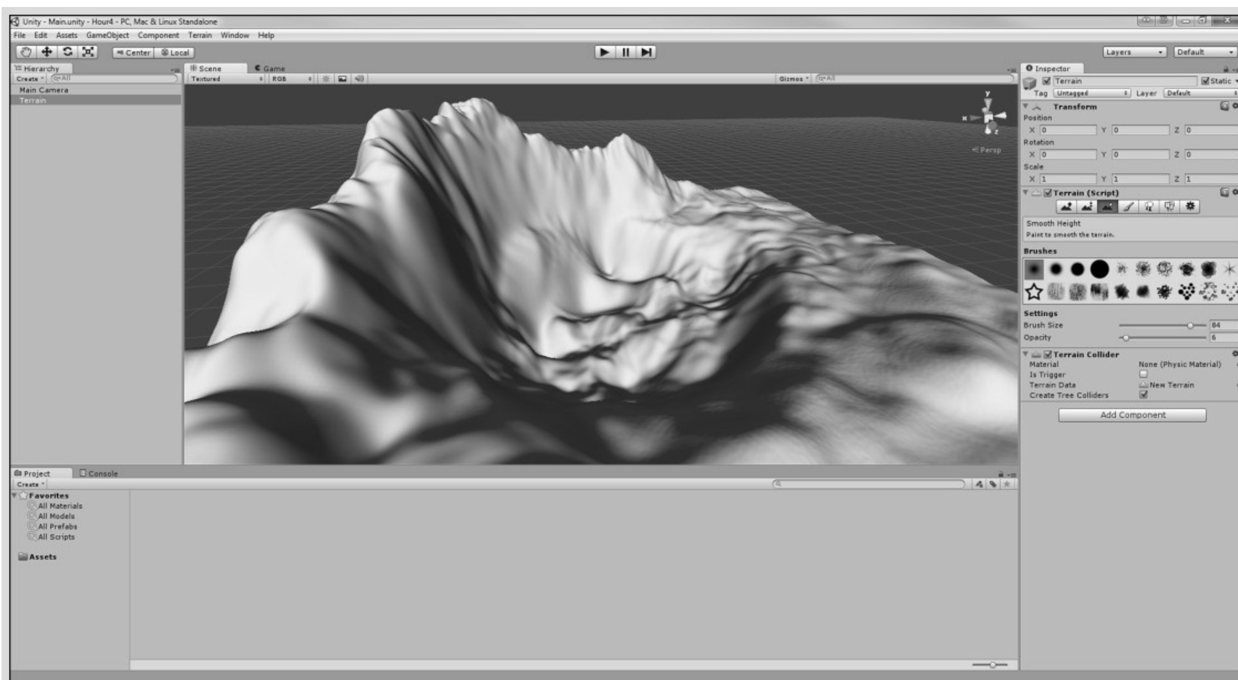


图4.7 示例地形

提示：

实践、实践、再实践

开发强大的、有吸引力的关卡本身也是一种艺术形式。必须对小山丘、峡谷、大山和湖泊的位置进行深思熟虑。这些元素不仅需要在视觉上令人满意，在放置它们时还需要考虑使关卡是能够是玩家过关的。这种技能不可能在一夜之间获得。一定要实践并精雕细琢你的关卡构建技能，以创建激动人心且令人难忘的关卡。

4.2 地形纹理

你现在知道如何构建 3D 游戏世界的物理维度。即使你的风景可能具有许多特色，它仍然是平淡无奇的并且难以导航。现在应该向关卡中添加一些角色。在本节中，你将学习如何绘制地形的纹理，给它提供一种迷人的外观。

4.2.1 导入地形资源

像雕刻地形一样，设置地形的纹理也非常像绘画。你选择一种画笔和纹理，并把它绘制到你的游戏世界中。不过，在可以开始利用纹理绘制游戏世界之前，需要一些可以使用的纹理。Unity 具有一些地形资源可供你使用，但是需要先导入它们。要加载这些资源，可以单击Assets > Import Package > Terrain Assets 命令，此时将出现Importing Package 对话框，如图4.8所示。在这个对话框中可以准确指定你希望导入哪些资源。如果你想减小项目的大小，那么取消选择不需要的项目就是一个好主意。目前，只需保持选中所有的选项即可，并单击Import 命令。你现在应该在Project 视图中的Assets 下具有一个名为Standard Assets 的新文件夹，这个文件夹包含你将在本章余下部分使用的所有地形资源。

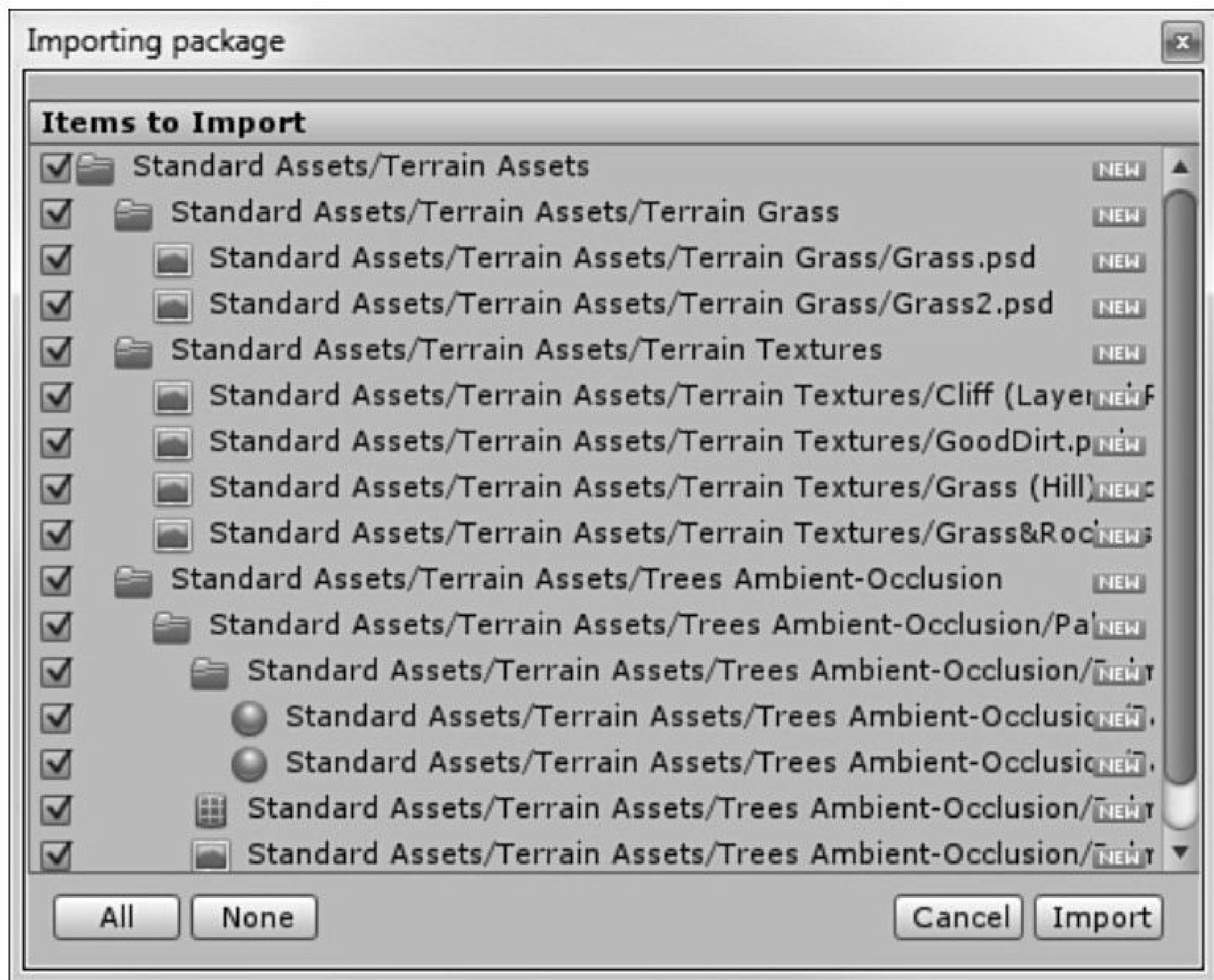


图4.8 Importing Package对话框

4.2.2 纹理化地形

地形纹理化过程在 Unity 中很简单，并且其工作方式与雕刻非常像。你需要做的第一件事是加载纹理。图4.9显示了Inspector中的纹理化工具。注意3个数字属性：画笔大小、不透明度和目标强度。你应该熟悉前两个属性，但是最后一个属性是新增的。目标强度是它通过持续绘画可以实现的最大不透明度。它的值是一个百分数，其中1就是100%。可以使用它作为一种控制方式，阻止过于强烈地绘制纹理。

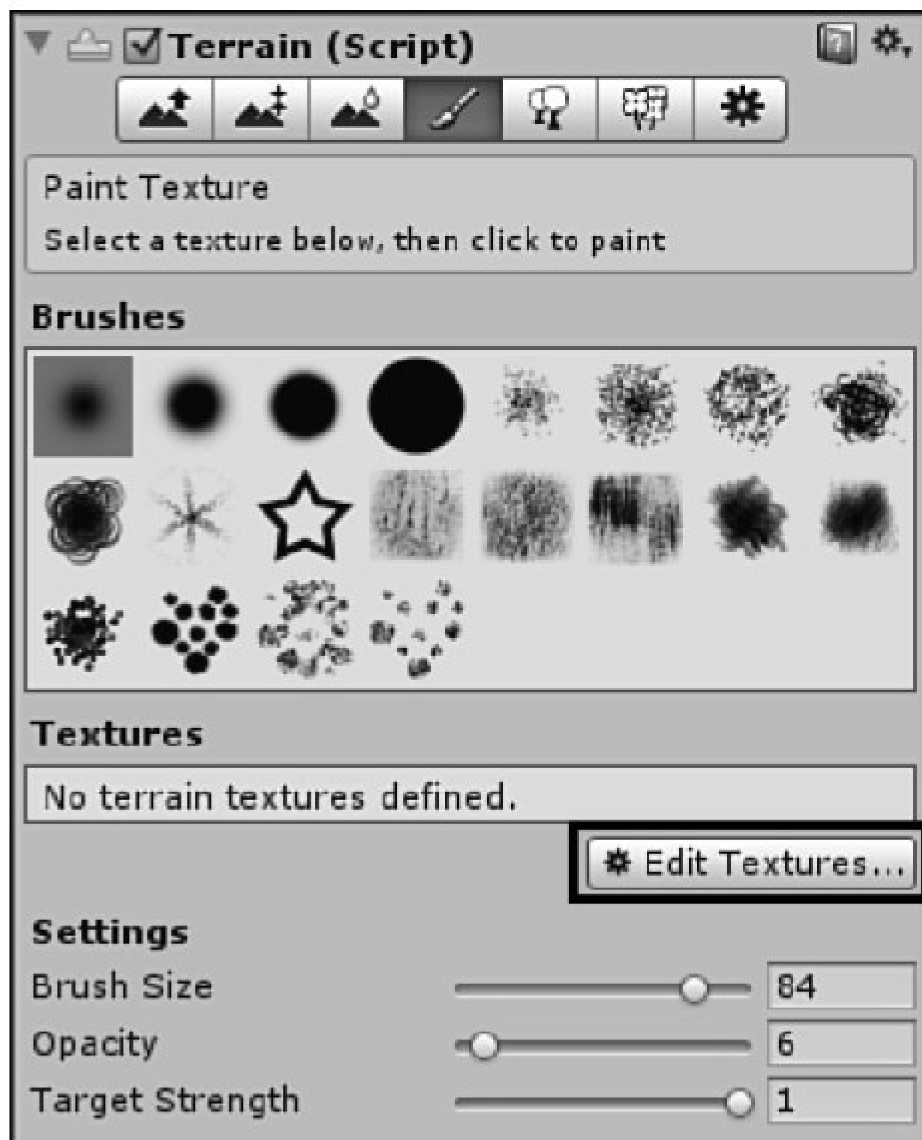


图4.9 地形纹理工具和属性

要加载一种纹理，可遵循下面这些步骤。

（1）单击Edit Textures > Add Texture 命令。

（2）出现Add Terrain Texture 对话框。单击Texture方框中的Select，如图4.10 所示，并选择Grass (Hill)纹理。

（3）单击Add按钮。

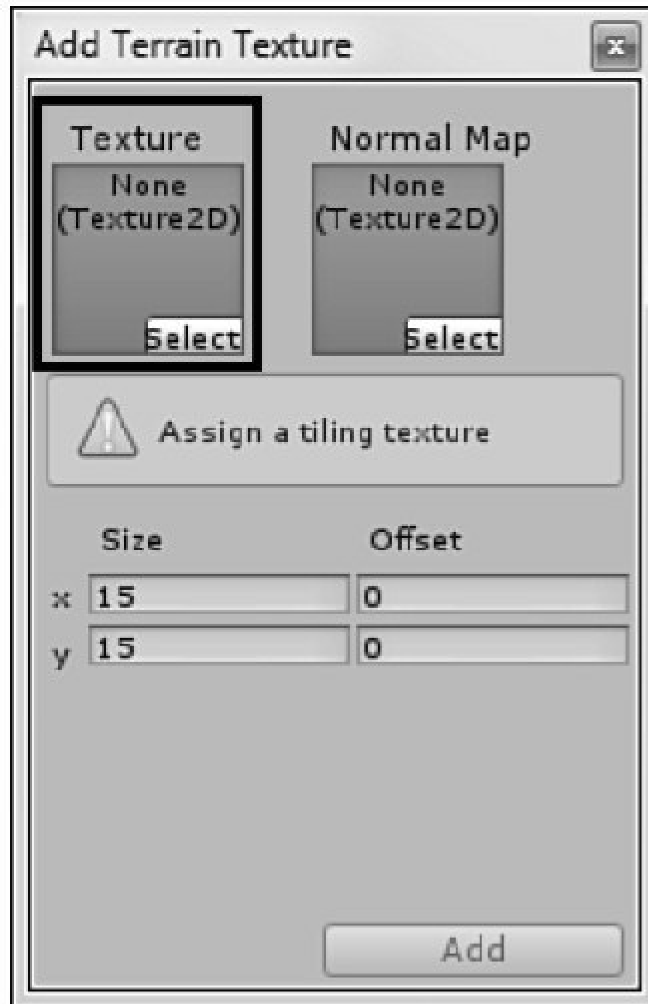


图4.10 Add Terrain Texture对话框

此时，整个地形都应该被东一块西一块的青草所覆盖。这看上去比以前的白色地形更好一些，但是仍然离逼真还很遥远。现在，你将实际地开始绘画，并使地形看上去更好。

在地形上绘制纹理

让我们对地形应用一种新纹理，给它提供一种更逼真的双色效果。

（1）使用前面列出的步骤，添加一种新纹理。这一次，加载Grass&Rock 纹理。一旦加载了它，就一定要通过单击以选择它（如果选择了它，在它下面就会出现一个蓝条）。

（2）把画笔大小、不透明度和目标强度分别设置为30、20和0.6。

（3）为节约起见，只在地形的陡峭部分和石缝上绘画（单击并拖动）。这会给人留下青草将不会在陡坡的两侧以及小山丘中间生长，如图4.11所示。

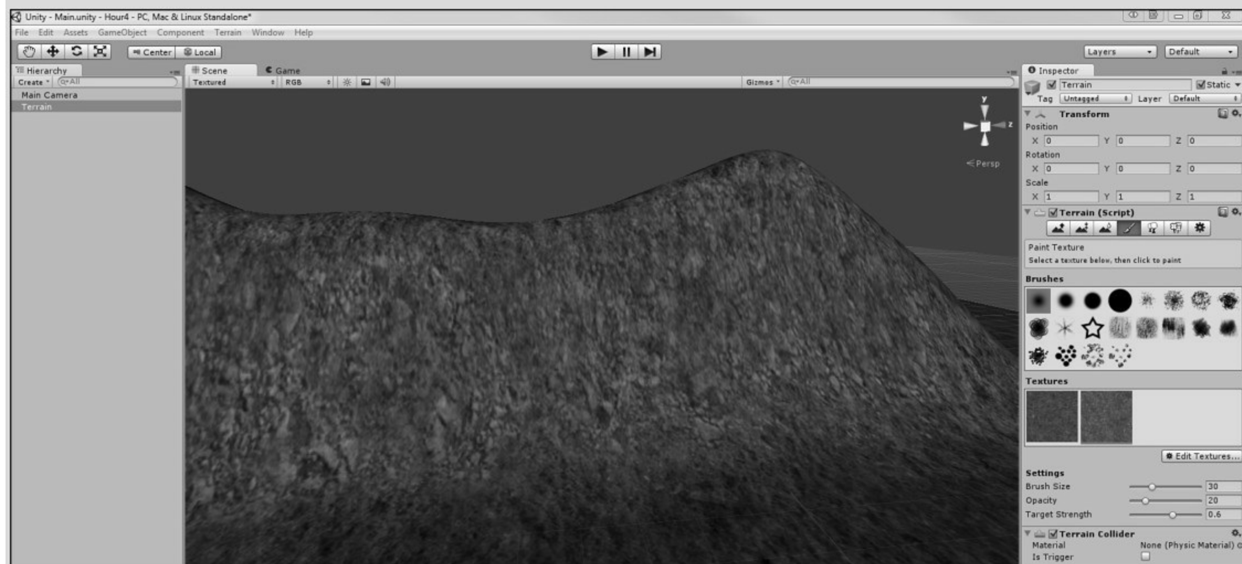


图4.11 双色的纹理化悬崖峭壁的示例

（4）继续试验纹理绘画。可以自由地加载纹理Cliff，并把它应用于更陡峭的部分，或者加载纹理Sand，创建一条小路。

你可以根据需要用这种方式加载许多纹理，并且实现一些逼真的效果。一定要实际练习纹理化，以确定看上去最好的图案。

注意：

创建地形纹理

游戏世界通常是独特的，需要自定义的纹理，以适应创建它们的游戏的环境。在为地形创建自己的纹理时，可以遵循一些一般的指导原则。第一条是总是要尽量使图案是可重复的，这意味着纹理可以无缝地拼贴。纹理越大，重复的图案就越不明显。第二条指导原则是使纹理成为正方形。最后一条指导原则是尽量使纹理尺寸是2的幂（32、64、128、512等）。最后两条指导原则会影响纹理的压缩和纹理的效率。利用一点实践，立刻就能创建出壮丽的地形纹理。

提示：

细微是最佳的策略

在纹理化时，记住使效果保持细微。在一个元素上体现出的大多数性质，在另一个元素上都会逐渐消失，而不会产生刺眼的渐变。你的纹理化工作也应该反映这一点。如果可以把摄像机拉离一块地形，并且指出一种纹理开始的准确位置，那你的效果就显得太突兀了。处理许多小的、细微的纹理应用比处理一个广泛的应用通常更好一些。

4.3 小结

在本章中，你认识了 Unity 中的地形。你首先学习了地形是什么，以及如何把它们添加到场景中。接着，你研究了利用高度图和 Unity 内置的雕刻工具雕刻地形。最后，你学习了如何通过以一种逼真的方式应用纹理，使地形看上去更有吸引力。

4.4 问与答

问：我的游戏必须具有地形吗？

答：根本不是。许多游戏完全是在建模的室内或者抽象的空间里发生的。

问：我的地形看上去不是非常好，这正常吗？

答：要花一些时间熟悉雕刻工具的使用。借助一些实践，关卡将开始看起来好得多。真正的质量来自于关卡测试，这将在下一章中介绍。

4.5 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

4.5.1 问题

1．判断题：在Unity地形中可以创建洞穴。

2．包含地形海拔信息的灰度图像被称为什么？

3．判断题：在Unity中雕刻地形非常像绘画。

4．怎样访问Unity可用的地形纹理？

4.5.2 答案

1．错误，Unity的地形不能交叠。

2．高度图。

3．正确。

4．通过选择Assets > Import Package > Terrain Assets命令，导入地形资源。

4.5.3 练习

让我们练习地形雕刻和纹理化。雕刻一种包含以下元素的地形。

湖床

海滩

山脉

平原

一旦你雕刻了这些元素，就以下面的方式对地形应用纹理。在

Terrain Assets 程序包中可以找到这里列出的所有纹理。

海滩应该使用Sand 纹理，并且应该淡入到Grass&Rock 中。

平原和所有平坦的区域都应该利用Grass 进行纹理化。

随着地形变得更陡峭，Grass 纹理应该平滑地渐变成Grass&Rock。

在到达最陡峭、最高的位置时，Grass&Rock 纹理应该渐变成Cliff。

用你想要的创意完成这个练习，构建一个让你自豪的游戏世界。

第5章 环境

在本章中你将学到：

怎样向地形中添加树木和青草；

怎样向地形中添加环境效果；

怎样利用角色控制器导航你的游戏世界。

在上一章中，你学习了为游戏雕刻和纹理化地形。在本章中将添加一些环境效果，它们将真正给你的游戏世界提供角色。你首先将学习如何向地形中添加像树木和青草这样的植物。接着，你将学习应用像水、天空、雾和镜头光晕（lens　flare）这样的效果。最后将向场景中添加一个角色控制器，并在你的游戏世界里四处走动。

5.1 生成树木和青草

只有平面纹理的游戏世界将会令人厌烦。几乎每一处自然的风景都具有某种形式的植物生命。在本节中，你将学习如何添加和自定义树木和青草，给地形提供一种有机的外观和感觉。

5.1.1 绘制树木

向地形中添加树木就像上一章中的雕刻和纹理化一样，整个过程与绘画非常相似。基本的前提是加载一个树木模型，设置树木的属性，然后绘制你希望树木出现的区域。基于所选的选项，Unity 将把树木分散开并且会改变它们，以提供一种更自然、更有机的外观。

提示：

地形资源

要学习本节余下的内容，需要把标准的地形资源加载进项目中。如果你的项目中尚没有它们，可以参阅上一章的内容，了解关于如何把它们导入到项目中的指导。

你将使用 Place Trees 工具把树木散布在地形上。一旦在场景中选择了地形，就可以在Inspector 视图中作为Terrain (Script)组件的一部分访问Place Trees 工具。图5.1 显示了Place Trees工具及其标准的属性。

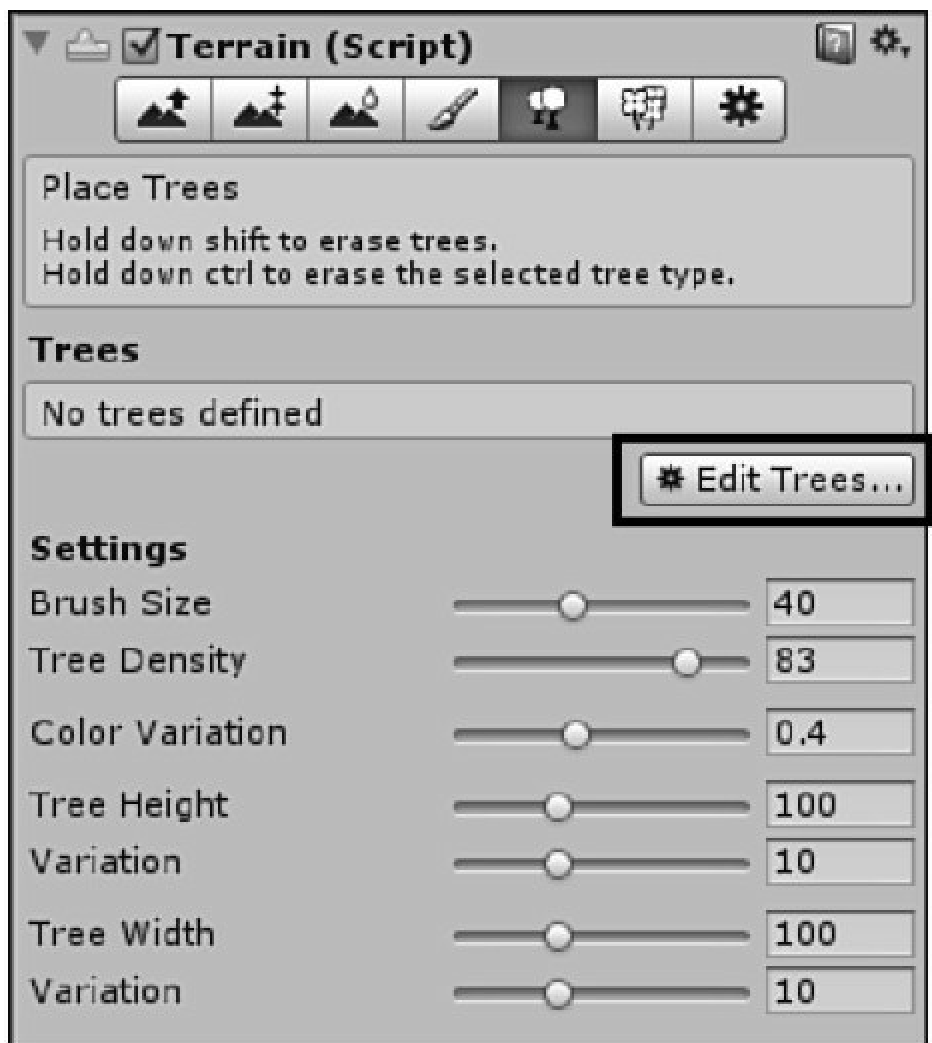


图5.1 Place Trees工具

表5.1 描述了Place Trees 工具的属性。

表5.1 Place Trees工具的属性

属性	描述
Brush Size	在绘画时将添加树木的区域的大小
Tree Density	树木能够多稠密地挤在一起
Color Variation、Tree Height/Width 和 Variations	这些属性允许所有的树木彼此稍有不同。这给人留下了许多不同树木（而不是重复同一棵树）的印象

在地形上放置树木

让我们执行下面这些步骤，使用Paint Trees 工具把树木放到地形上。这个练习假定你创建了一个新场景，并且已经添加了地形。应该把

地形的长度和宽度都设置为 100。如果地形已经完成了某种雕刻和纹理化工作，它看起来将会更好。

（1）单击Edit Trees > Add Tree 命令，调出Add Tree 对话框，如图5.1所示。

（2）单击Add Tree 对话框上的Tree 文本框右边的圆形图标，调出Tree Selector 对话框，如图5.2所示。



图5.2 Add Tree对话框

（3）选择Palm Tree，并单击Add 按钮。

（4）把画笔大小设置为10，把树密度设置为70，并把宽度和高度设置为50。选择你想要的任何变化属性。

（5）在希望出现树木的区域上单击并拖动，在地形上绘制树木。在单击并拖动时按住Shift键，将会删除树木。

（6）继续试验不同的画笔大小、密度以及树木大小/变化。

注意：

黑暗的树木

你可能注意到一些树木看上去是黑色的。当把较小的树放在较大的树附近时，就可能发生这种情况。其原因是：无论场景中存在任何光照，都会计算为触碰小树周围的大树。实际上，小树处于大树的阴影中。可以利用Unity Pro解决这个问题，因为光照是动态计算的，并且倾向于更准确一点。如果你没有Pro版本，也可以简单地删除并替换树木。

注意：

树木弯曲

在Scene视图中四处移动时，可能会看到一些树木弯曲了并且在改变。你见证到的是使项目运行得更快的内置效率。当树木远离观众时，它们将被渲染为质量低得多的公告板（billboard）。当观众靠近时，树木将弯曲成更高质量的版本。其效果是你所看到的弯曲形状。你可以通过修改一些地形设置，更改这些渐变何时以及如何发生。在本章后面将有机会处理这些设置。

5.1.2 绘制青草

既然你已经学习了如何绘制树木，现在应该学习如何给游戏世界应用青草或其他小植物生命。青草或其他小植物在Unity中称为细节（detail）。因此，用于绘制青草的工具是Paint Details工具。与树木（它们是3D模型）不同，细节是公告板（后面将加以解释）。就像你到目前为止反复看到过的，使用画笔和绘画运动将细节应用于地形。图5.3 显示了Paint Details工具以及它的一些属性。

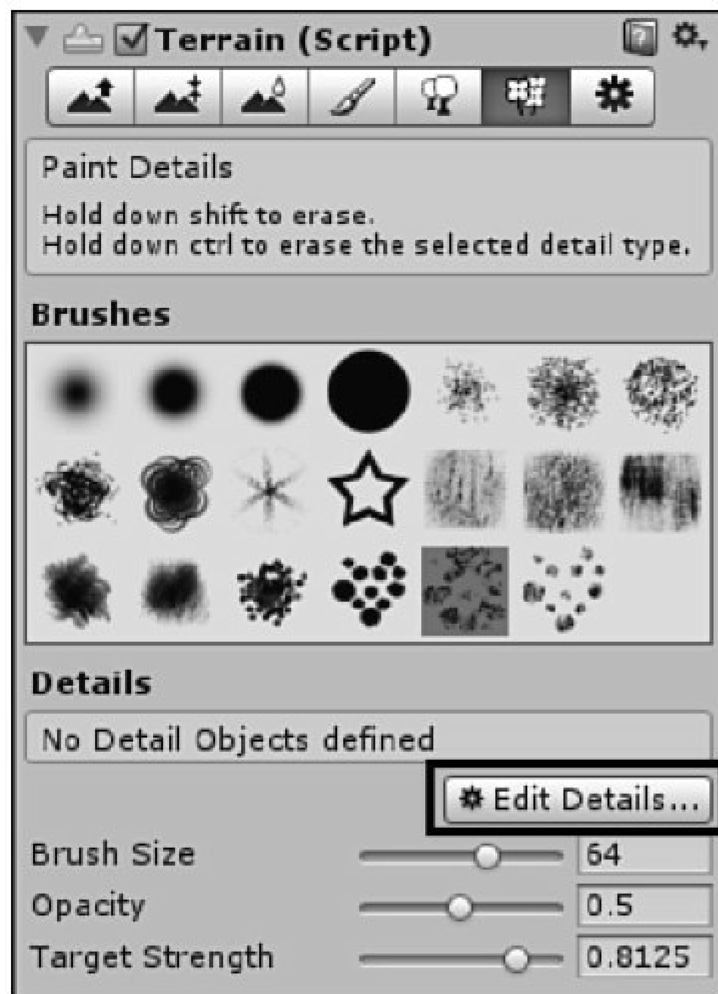


图5.3 Paint Details工具

注意：

公告板

公告板是3D游戏世界中的一种特殊的可视化组件，它可以提供3D模型的效果，但是不必实际地成为一种3D模型。模型存在于全部3个维度中，因此，在围绕其中一个维度运动时，可以看到不同的侧面。不过，公告板是始终面向摄像机的平面图像。当尝试转动公告板时，它将转而面对你的新位置。公告板常用于青草细节、粒子和屏幕上的效果。

将青草应用于地形是一个相当直观的过程，首先需要添加青草纹理，其操作如下。

（1）在Inspector视图中单击Edit Details，并选择Add Grass Texture

命令。

（2）在Add Grass Texture对话框中，单击Texture 文本框旁边的圆形图标，如图5.4 所示，并选择Grass 纹理，而不是Grass (Hill)纹理。

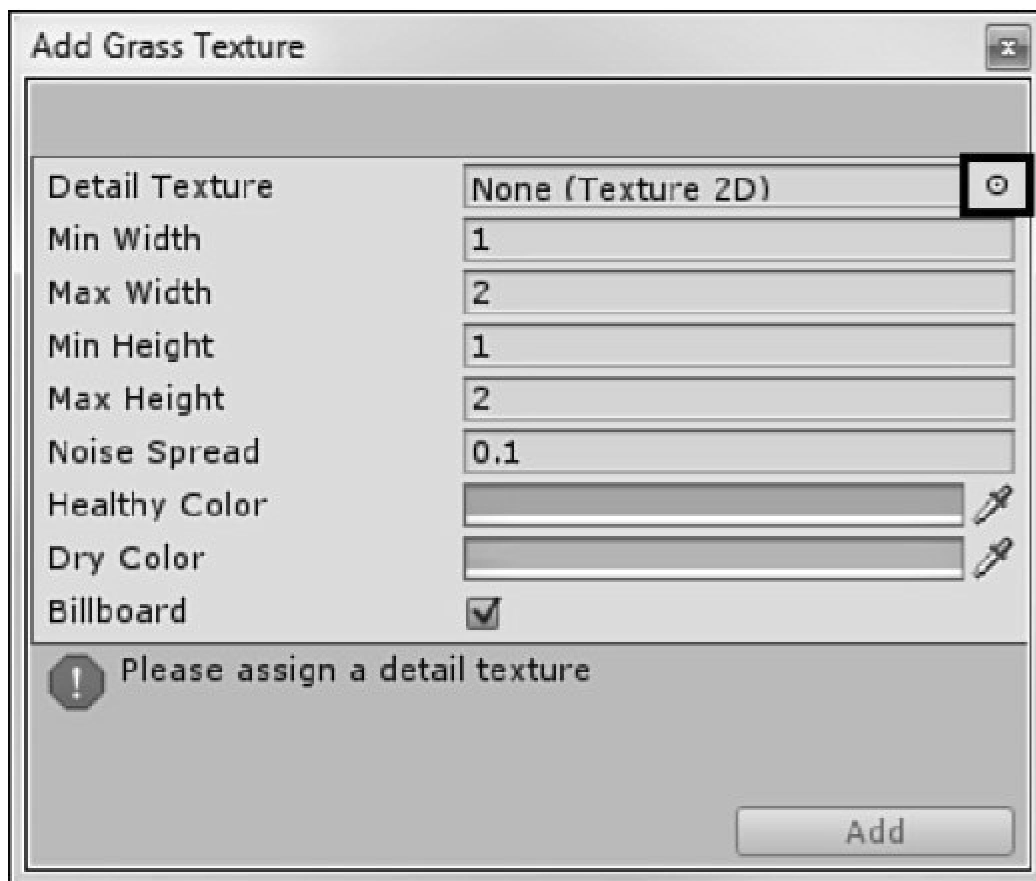


图5.4 Add Grass Texture对话框

（3）把纹理属性设置为你想要的任何值。要特别注意颜色属性，因为它们将为青草建立自然颜色的范围。

（4）完成后，单击Apply按钮。

在加载了青草之后，只需选择一种画笔和画笔属性。现在就准备好开始绘制青草。

提示：

逼真的青草

你可能注意到：在开始绘制青草时，它看上去并不逼真。在把青草

添加到地形中时，你需要重点关注以下几件事。第一件是注意为青草纹理设置的颜色。要尽量使它们的颜色更深、更具土色调。需要做的下一件事是选择一种非几何的画笔形状，以便帮助打破硬边缘（如图5.3所示，了解要使用的良好画笔）。最后，保持透明度和目标强度这些属性非常低，其中每个属性的良好的起始设置是0.02。如果需要更多的青草，可以只在相同的区域上绘画。

警告：

植物和性能

场景中具有的树木和青草越多，渲染它所需的处理也越多。如果你关注的是性能，就需要保持较低的植物数量。在本章后面将探讨一些属性，它们有助于管理这些，但是作为一条简单的规则，要尽量只在那些确实需要的区域添加树木和青草。

提示：

消失的青草

与树木一样，青草也会受到它与观众之间的距离影响。树木在观众远离时将恢复较低的质量，而青草完全不会渲染出来。结果就是在不能看到青草的观众周围显示一个圆环。同样，可以通过将在本章后面探讨的属性来修改这个距离。

5.1.3 地形设置

Inspector视图中的这个地形工具列表上的最后一个按钮用于Terrain Settings工具。这些设置从总体上控制地形、纹理、树木和细节的外观和工作方式。图5.5显示了所有的地形设置。

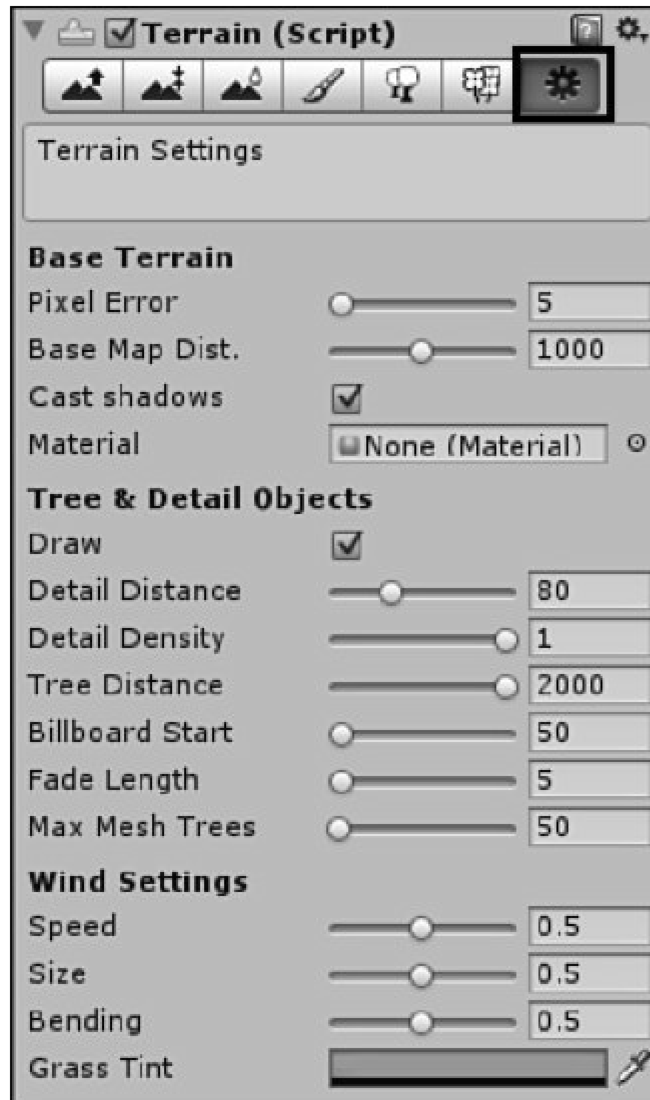


图5.5 Terrain Settings 工具

第一组设置用于总体的地形。表5.2描述了多种设置。

表5.2 基本的地形设置

设置	描述
Pixel Error	在显示地形的几何形状时所允许的错误数量。这个值越高，地形的细节越低
Base Map Dist.	将显示高分辨率纹理的最大距离。当观众比给定的距离更远时，纹理将降级为更低的分辨率
Cast Shadows	确定地形的几何形状是否会投射阴影
Material	这个属性用于指定能够渲染地形的自定义材质。材质必须包含能够渲染地形的着色器

此外，一些设置会直接影响树木和细节（比如青草）在地形中的表现方式。表 5.3 描述了这些设置。

表5.3 树木和细节对象设置

设置	描述
Draw	确定是否在场景中渲染树木和细节
Detail Distance	将不再把细节绘制到屏幕上的摄像机距离
Tree Distance	将不再把树木绘制到屏幕上的摄像机距离
Billboard Start	3D 树木模型将开始渐变成低质量公告板的摄像机距离
Fade Length	树木将在公告板与高质量 3D 模型之间渐变的距离。这个设置的值越高，渐变就越平滑
Max Mesh Trees	能够同时绘制为 3D 网格（而不是公告板）的树木总数

将要探讨的最后几个设置是用于风的。由于你还没有机会实际地在你的游戏世界里逛逛（在本章后面将会这样做），你可能想知道它们意味着什么。基本上，Unity会在你的地形上面模拟一种微风，它将导致青草弯曲和摇摆，并使游戏世界具有生气。表5.4描述了风的设置。

表5.4 风的设置

设置	描述
Speed	风效果的速度以及强度
Size	同时被风影响的青草区域的大小
Bending	青草由于风而将具有的摇摆程度
Grass Tint	尽管不是一种风设置，但是该设置可以控制关卡中的所有青草的总体色调

5.2 环境效果

至此，你已经雕刻和纹理化了地形，并向其中添加了树木和青草。可以安全地说，它看上去比只是平面的白色正方形要好得多。在本节中，你将学习添加环境细节，真真切切地使游戏世界尽可能完善。

5.2.1 天空盒

你可能注意到：虽然地形中充满了纹理和细节，但是天空是柔和的纯色。你需要做的是给游戏世界添加一个天空盒，它是一个包围游戏世界的大盒子。即使它是由6个平面状的侧面组成的立方体，它也具有面朝里的纹理，使得它看上去像无边无际的苍穹。你可以创建自己的天空盒，或者使用Unity的标准天空盒之一。在本书中，你将使用内置的天空盒。

要使用标准天空盒，需要把资源导入到项目中。要导入天空盒，可以单击Assets > Import Package > Skyboxes 命令。这将打开Import Package 对话框。保持选中所有的选项，并单击Import按钮。在导入资源之后，就可以开始处理天空盒。

可以用两种方式把天空盒添加到游戏世界中：其一是把天空盒添加给摄像机；另一个是把它添加到场景中。

5.2.2 把天空盒添加给摄像机

可以把天空盒添加给摄像机，使得摄像机在游戏世界之外看到的任何内容都将被天空所取代。要把天空盒添加给摄像机，可以遵循下面这些步骤。

（1）在Hierarchy视图中选择Main Camera。

（2）单击Component > Rendering > Skybox 命令，添加一个天空盒组件。

（3）在Inspector视图中，定位Skybox组件，并单击Custom Skybox框旁边的圆形图标，如图5.6 所示。在Select Material 对话框中，选择Sunny2 Skybox。

（4）运行场景，查看应用于摄像机的天空盒。

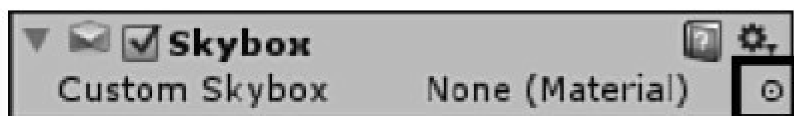


图5.6 Skybox组件

注意：

多个天空盒

有一个选项用于把天空盒添加给特定的摄像机，提供这个选项的原因是使你可以在不同的摄像机上具有不同的天空盒（或者没有天空盒）。这使你能够给不同的观众显示不同的游戏世界。如果希望能够灵活地观察游戏世界，就要把天空盒添加给摄像机。如果希望游戏世界对任何人看上去都是一致的，就要把它添加到场景中（下面将介绍）。

5.2.3 把天空盒添加到场景中

如果把天空盒添加到场景中，它对于所有的观众都将是存在的。这种方法的另一个好处是：天空盒在Scene视图中是可见的，这使得很容易查看游戏世界在具有所有元素的情况下是什么样子的。要把天空盒添加到场景中，可以遵循下面这些步骤。

（1）单击Edit > Render Settings 命令，在Inspector视图中打开场景的渲染设置。

（2）定位Skybox Material框，并单击它右边的圆形图标。

（3）选择Sunny1　Skybox。注意Scene视图如何变化以包含天空。如果Scene　没有变化，可以启用天空盒、雾和镜头光晕场景设置，如图5.7所示。



图5.7 环境效果切换

5.2.4 雾

在 Unity 中，可以把雾添加到场景中，并使用这种雾来模拟许多不同的自然现象，比如薄雾、实际的雾或者远距离的对象衰减。还可以使用雾给游戏世界提供新的异域风情。

把雾添加到场景中

让我们把雾添加到场景中，并且学习影响它的不同属性。

（1）单击Edit > Render Settings 命令，将在Inspector视图中打开渲染设置。

（2）选中Fog复选框，启用雾。

（3）试验不同的雾浓度和颜色。表5.5描述了多种雾属性。

有多个属性会影响雾在场景中的外观，表5.5描述了这些属性。

表5.5 雾属性

设置	描述
Fog Color	雾效果的颜色
Fog Mode	它用于控制计算雾的方式。3 种模式是 Linear、Exponential 和 Exp2。Linear 将是平滑的雾渐变，并且是默认模式
Fog Density	雾效果有多强烈。仅当把雾模式设置为 Exponential 或 Exp2 时，才会使用这个属性
Linear Fog Start Linear Fog End	这些用于控制雾开始时离摄像机有多近，以及它结束时离摄像机有多远。这些属性只在 Linear 模式中使用

5.2.5 镜头光晕

无论何时摄像机查看明亮的光源，都会发生镜头光晕这种视觉畸变。它是灯光在镜头的玻璃内反射的结果。在尝试观看像太阳这样的明亮光源时（不建议这样做），也可能会体验到镜头光晕。在 Unity 中，可以把光晕添加到光源中，给它们提供一种更逼真的效果，并使得它们看上去似乎非常明亮一样。

向场景中添加镜头光晕

如果遵循下面逐步的指导，将更容易查看如何将镜头光晕放置到场景中。添加光晕非常简单，但它要使用一些你可能还不熟悉的新项目。在可以向场景中添加光晕之前，需要具有一个光源和一些光晕资源，在下面的步骤中将把它们都料理好。

（1）单击GameObject > Create Other > Directional Light 命令，向场景中添加一种定向灯光。在后面的一章中将更详细地介绍定向灯光。目前，只需了解定向灯光是一种平行灯光，就像太阳一样。

（2）一旦向场景中添加了灯光，就可以旋转它，使之在地形上提供想要的灯光效果。

（3）接下来，需要光晕资源。可以单击Assets > Import Package > Light Flares 命令，导入Unity光晕资源。在Import对话框中，保持选中所有的选项，并单击Import按钮。

（4）在Hierarchy视图中选择定向灯光，并在Inspector视图中定位Flare属性。

（5）单击Flare 属性旁边的圆形图标，并从Select Flare 对话框中选择Sun光晕。

此时，光晕就出现在灯光上，并将被摄像机获取。这是由于场景的Main Camera 默认具有Flare Layer 组件。没有该组件的任何摄像机都将不能看到镜头光晕。

提示：

什么是光晕？

你也许还不能看到镜头光晕，因为你的摄像机没有指向定向灯光。此时不要麻烦地尝试使摄像机指向灯光。在本章后面，你将在场景运行时四处移动它。到那时，你将能够看到它，并且执行任何需要的调整。

5.2.6 水

你将考虑添加的最后一种环境效果是水。水是一种将依据你是具有Unity Pro 版本还是免费版本而改变的效果。Unity 的免费版本能够访问基本的水。这个对象有点普通，但是还过得去。水的Pro版本看上去要好得多。如果你具有Pro版本，那么你肯定会选择这个版本。由于本书被编写成使用免费版本，那就使用这个版本好了。

在 Unity 中，水是一种需要导入的资源。在场景中，水是一种平坦的平面，看起来就像池塘或湖泊的顶部表面区域。注意：水只是一种效果。如果玩家跳入有水的湖泊中，玩家将会在水中坠落，并下潜到为它雕刻的孔中。

创建湖泊并添加水

要添加水，需要地形的某个部分包含水。在这个练习中，将雕刻一个湖泊并向其中添加一些水。

（1）创建一种新地形或者处理现有的地形。在地形中向下雕刻进一个湖床。

（2）单击Assets > Import Package > Water (Basic)命令，导入水资源。在Import Package对话框中，保持选中所有的选项，并单击Import按钮。

（3）在Project 视图中定位Water (Basic)文件夹，并且定位Daylight Simple Water资源，如图5.8所示。

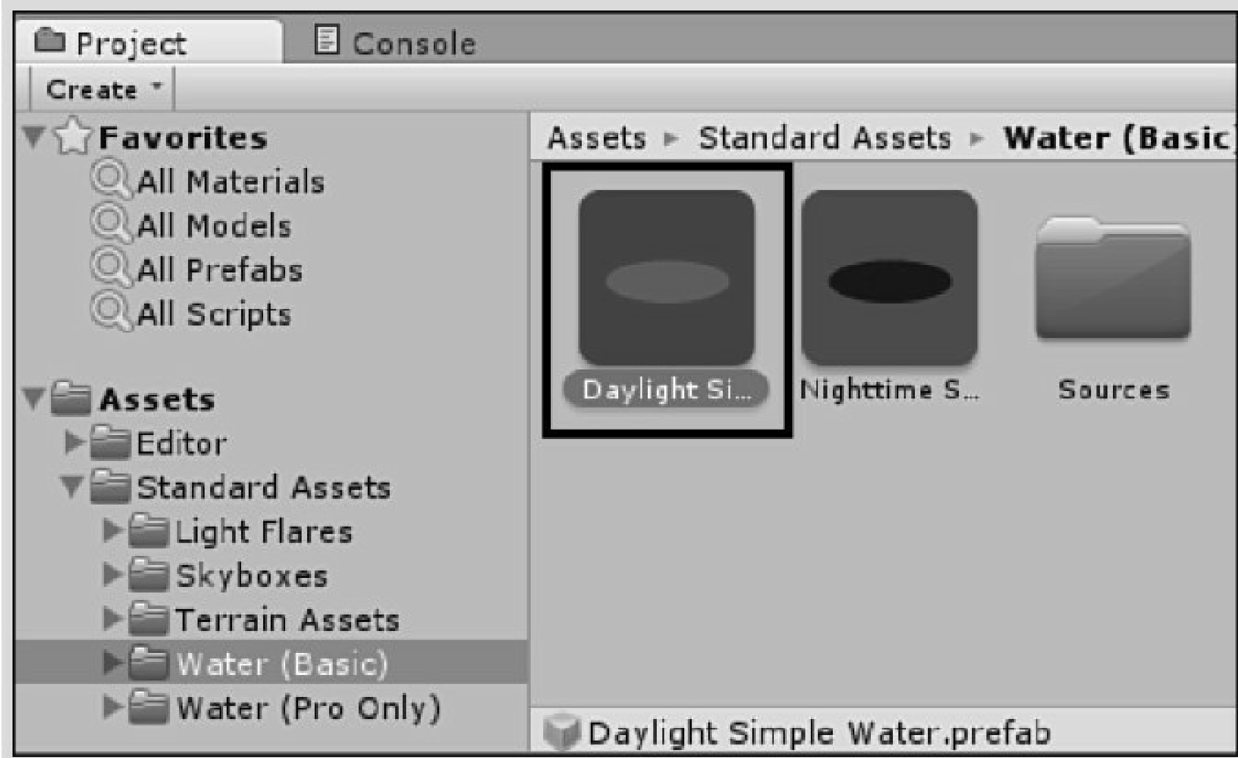


图5.8 Water (Basic)文件夹和资源

（4）把Daylight Simple Water资源拖到Scene 视图上以及你创建的湖床中。根据需要缩放和移动水，使之正确地适应湖床。

5.3 角色控制器

此时，你已经完成了地形。你雕刻和纹理化了地形，给它添加了树木和青草，并提供了一个天空，并且它还具有雾、镜头光晕和水。现在应该进入关卡，开始“玩”游戏。Unity提供了两种基本的角色控制器，使你自己无需做许多工作即可轻松地进入场景中。实质上，你是把一个控制器放入场景中，然后利用大多数第一人称游戏常用的控制模式四处移动。

5.3.1 添加角色控制器

要向场景中添加角色控制器，首先需要导入资源。单击 Assets > Import Package >Character Controller 命令，在Import Package 对话框中，保持选中所有的选项，并单击Import按钮，应该会把一个名为 Character Controllers 的新文件夹添加到 Project 视图中的 Standard Assets文件夹下。由于你没有将用作玩家的3D模型，我们将使用第一人称控制器。在Character Controllers 文件夹中定位First Person 控制器资源，如图5.9所示，并把它拖到Scene 视图中的地形上。

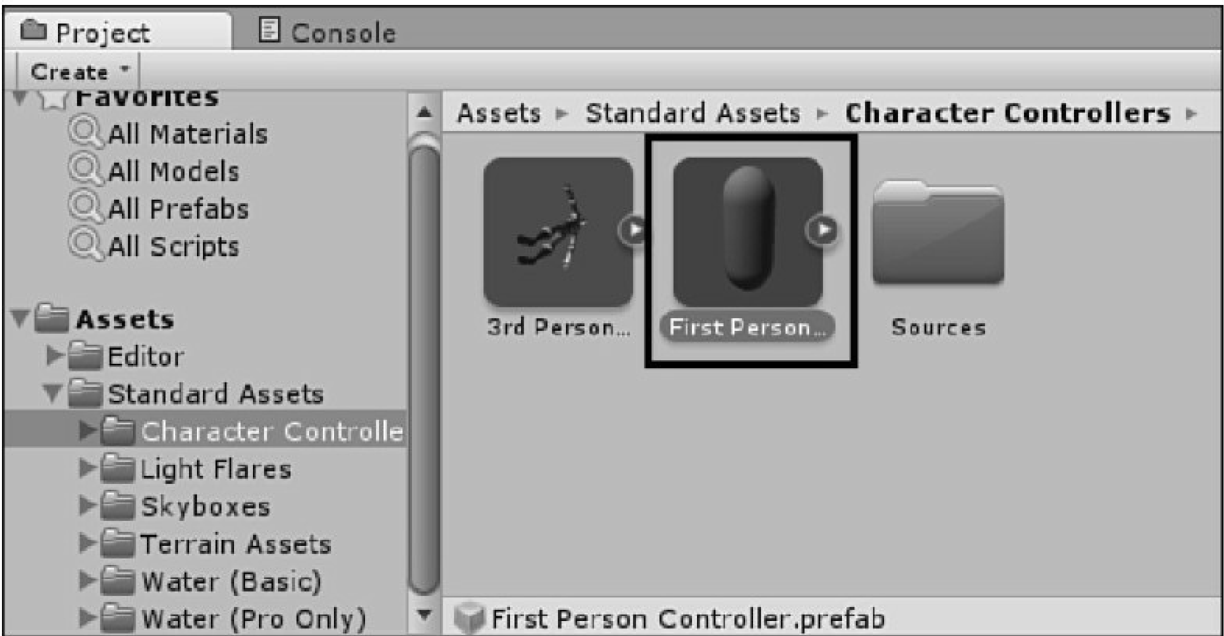


图5.9 First Person 角色控制器

既然已经把角色控制器添加到了场景中，就可以在你创建的地形中四处移动。在场景中玩游戏时，你将注意到现在可以从放置控制器的位置查看游戏世界。可以使用 W、A、S 和D键四处移动，使用鼠标观察周围的情况，以及使用空格键跳跃。如果你还不能得心应手地使用这些控制，就要不断试验它们，并且享受你在游戏世界里的体验！

提示：

“2个音频侦听器”

在把角色控制器添加到场景中时，你可能在编辑器底部注意到一条消息，指示“场景中有2个音频侦听器”。这是由于 Main Camera（默认存在的摄像机）具有一个音频侦听器组件，而你添加的角色控制器也是如此。由于摄像机代表玩家的视角，只有一个能侦听音频，因此可以通过从 Main Camera 中删除音频侦听器组件来修正这个问题。

提示：

从游戏世界里坠落

无论何时你运行场景，如果发现摄像机从游戏世界里坠落，这可能

是由于角色控制器部分陷入地面下所致。尝试把角色控制器抬高一点，使之位于地面之上。当场景开始运行时，摄像机应该只会下落一点点，直至它碰到地面并停止下落。

注意：

导入资源

在本章中，你导入了许多资源程序包。在导入它们时，保持选中Import Package对话框中的所有选项。这导致把该程序包中的所有资源都添加到项目中。这样做可能会使项目文件变得非常大。在更现实的情况下，只会导致需要使用的资源。这样，将会取消选中所有不需要的资源。记住，以后总是可以根据需要导入它们！

5.3.2 修正游戏世界

既然你已经可以进入游戏世界并且近距离观察它，现在就应该细化一些较小的细节。你可能注意到一些构建为小路的区域过于陡峭，以至于无法行走，还可能会看到一些区域上的纹理放置得不尽合理。现在应该消除游戏世界里的所有障碍，并修正你发现的任何错误。从Scene 视图中可能难以查看所有需要修正的位置，直到你实实在在地在游戏世界里四处移动时，才真正有机会体验它。

一种值得探讨的修正方法是前面添加的镜头光晕。如果仰望天空，就会注意到太阳是天空盒纹理的一部分。你还可能会注意到镜头光晕。模拟太阳的镜头光晕与太阳图像本身可能没有对齐。在查看太阳和光晕时，可以暂停场景。注意：在查看特定的事物时，可能难以利用鼠标暂停场景。在场景运行时，可以按下Shift+Ctrl+P组合键（在Mac上则是Shift+Cmd+P组合键）轻松地暂停场景。在Scene视图中，可以旋转定向灯光，直到太阳图像与光晕对齐为止。要记录下旋转信息，因为一旦你停止场景，任何更改都会丢失。一旦停止了场景，只需简单地把定向灯

光旋转回与太阳图像对齐的方向即可。这样，就可以从你的列表中划掉另一个细节。

5.4 小结

在本章中，你学习了 Unity 中的环境细节。首先学习了向场景中添加树木和青草，然后添加了一些环境效果，比如天空、雾和镜头光晕。接着，你使用了 Unity 的水资源。在本章最后，你向场景中添加了一个角色控制器，并且实际地逛了逛你的游戏世界。

5.5 问与答

问：树和草会极大地影响性能吗？

答：这依赖于场景上同时具有它们的数量，还依赖于运行它的计算机的处理器能力。一个良好的规则是：如果树木和青草对场景具有积极的影响，就添加它们。

问：我可以创建自己的天空盒吗？

答：的是，你可以。如果你正在构建自定义的游戏世界或者具有可用天空盒中不存在的特定细节的游戏世界，通常应该考虑使用这种方法。

问：角色控制器有许多属性，我需要一一了解它们吗？

答：不需要。角色控制器很容易使用。如果需要简单地修改运动，可以那样做，但在大多数情况下应该不必如此。

5.6 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

5.6.1 问题

1．哪种设置用于控制树木在风中的摇摆幅度？

2．可以模拟远处的薄雾或颜色衰减的环境效果的名称是什么？

3．这个对象是一个适应游戏世界的立方体，将对其进行纹理化，使之看起来像天空。

4．你向场景中添加的是哪种角色控制器：First Person（第一人称）还是Third Person（第三人称）。

5.6.2 答案

1．这是一个具有欺骗性的问题。树木不会在风中摇摆，青草则不然。控制它的设置是Bending（在Wind Settings 下）。

2．雾。

3．天空盒。

4．First Person。

5.6.3 练习

在这个练习中，你有机会完成在第4章末尾开始创建的地形。你将添加其余的环境效果，给游戏世界提供更好的逼真度。

打开具有你在第4章中创建的地形的项目或场景，需要向其中添加以下内容。

给你创建的湖床添加水。

在湖床边缘周围添加一些稀草，并在平坦的平原上添加更多的草。

在长草的区域添加一些棕榈树，其中青草纹理将接触到海滩的沙子纹理。

向场景中添加一个天空盒。

向场景中添加一种雾效果。更改设置，使得它逼真地使山脉看上去像乌云密布。

添加一种定向灯光，并向灯光中添加镜头光晕。如果天空盒中存在太阳图像，确保灯光与之对齐。

添加角色控制器和测试来驱动关卡，确保一切放置正确并且看上去比较逼真。

第6章 灯光和摄像机

在本章中你将学到：

在Unity中怎样处理灯光；

摄像机的核心元素；

在场景中怎样处理多个摄像机；

怎样处理图层。

在本章中，你将学习如何在Unity中使用灯光和摄像机。首先将探讨灯光的主要特性，然后探索不同类型的灯光以及它们的独特应用。一旦学完了灯光，就开始处理摄像机。你将学习如何添加新的摄像机，放置它们，以及利用它们生成有趣的效果。最后将学习在Unity中处理图层。

6.1 灯光

在所有形式的视觉媒体中，灯光在其视觉感受方面都走了一段很长的路。明亮的淡黄色灯光可以使场景看上去很阳光、很温暖。同样的场景，给它提供一种强度较低的蓝色灯光，这时它看上去就有些怪异恐怖并且令人不安。大多数努力实现逼真或显著效果的场景都会寻求至少一种（通常是许多种）灯光。在过去，你只是简单地使用灯光以照亮其他元素。在本节中，将更直接地处理灯光。

注意：

重复属性

不同的灯光共享许多相同的属性。如果某种灯光具有已经在另一种灯光下介绍过的属性，那么将不会再次介绍它。只需记住：如果两种不同的灯光具有同名的属性，那么这些属性将会做相同的事情。

注意：

什么是灯光?

在Unity中，灯光本身并不是对象。相反，它们是组件。这意味着当把灯光添加到场景中时，实际上只是利用Light组件添加了一个游戏对象。这个灯光组件可以是你能够使用的任何类型的灯光。

6.1.1 点光源

你要处理的第一种灯光是点光源，可以把它视作一只灯泡。所有的灯光都是从一个中心位置向各个方向上发出的。点光源也是用于照亮内部区域的最常见的灯光类型。

要把点光源添加到场景中，可以单击GameObject > Create Other >

Point Light命令。一旦它位于场景中，就可以像其他任何对象一样操纵点光源游戏对象。表6.1描述了点光源的属性。

表6.1 点光源的属性

属性	描述
Type	Type 属性是组件发出的灯光类型。由于这是一个点光源，类型应该是 Point。更改 Type 属性将会改变灯光的类型
Range	Range 属性指示灯光照射的距离有多远。亮度将从光源到指示的范围均匀地衰减
Color	光照的颜色。颜色会进行加色，这意味着如果把红色灯光照在蓝色对象上，它最终将是紫色的
Intensity	Intensity 属性指示灯光照射的亮度。注意：灯光只能照射到 Range 属性所指示的距离
Cookie	Cookie 属性接受一个立方图（比如天空盒），指示灯光照射的模式。在后面将更详细地介绍 Cookie
Shadow Type	Shadow Type 属性是指如何为场景中的这个源计算阴影。硬阴影更精确，并且更加是性能密集的。所有的阴影都需要 Unity Pro 才能正常工作。如果使用 Unity Free，那么具有阴影的唯一方式是手动把它们烘焙进纹理中
Draw Halo	Draw Halo 切换选项用于确定在灯光周围是否会出现发光的晕轮。后面将更详细地介绍晕轮
Flare	Flare 属性接受灯光光晕资源，并且会模拟射入摄像机镜头里的亮光的效果。在前几章中你使用了灯光光晕来模拟太阳的效果
Render Mode	Render Mode 属性确定此灯光的重要性。3 个设置是 Auto、Important 和 Not Important。重要的灯光将以更高的质量渲染，而不太重要的灯光则会更快速地渲染
Culling Mask	Culling Mask 属性确定哪些图层会受灯光影响。默认情况下，所有的一切都会受灯光影响。后面将详细介绍图层
Lightmapping	Lightmapping 属性确定灯光是实时计算的，还是烘焙进灯光贴图中的。这是一个更高级的设置，目前还不需要它。只需保持将其设置为 Auto 即可

注意：

烘焙

烘焙（baking）是指在创建期间给纹理和对象添加灯光和阴影的过程。可以利用Unity或者图形编辑器执行该操作。例如，如果你将要利用与人影类似的暗斑制作一种墙面纹理，然后把一个人体模型放在墙面旁边，看起来就好像该模型正在向墙上投射阴影。不过，事实是阴影是烘焙进纹理中的。烘焙可以使游戏的运行速度更快，因为引擎将不必逐帧计算灯光和阴影。这是一个非常好的主意！

向场景中添加点光源

让我们构建一个具有一些动态点光源的场景。在本书配套资源中的Hour 6下提供了这个项目的完成版本，即Hour6_PointLight。

（1）创建一个新的场景或项目。

（2）向场景中添加一个平面（单击GameObject > Create Other > Plane 命令）。确保将该平面定位于(0, .5, 0)，并将其旋转(270, 0, 0)。这个平面对于摄像机应该是可见的。

（3）向场景中添加两个立方体，把它们分别定位于(−1.5, 1, −5)和(1.5, 1, −5)。

（4）向场景中添加一个点光源（单击GameObject > Create Other > Point Light 命令），并把该点光源定位于(0, 1, −5)。注意灯光如何照亮立方体的内侧和背景平面，如图6.1所示。

（5）继续探索灯光属性，一定要试验灯光颜色、范围和强度。

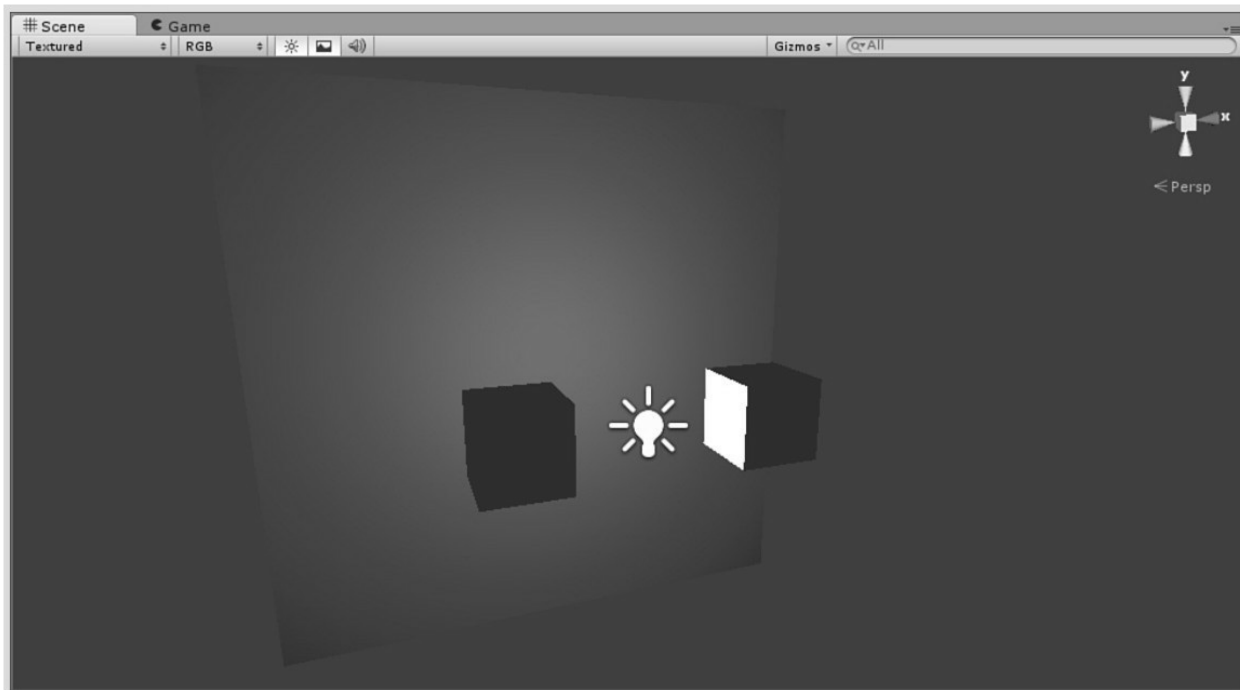


图6.1 练习的结果

6.1.2 聚光灯

聚光灯的工作方式非常像汽车的前灯或手电筒。聚光灯的光线开始于一个中心点，然后以圆锥体的形式发出光。换句话说，聚光灯将照亮

它们前面的一切对象，而将保持所有其他的一切处于黑暗中。与在每个方向上发光的点光源不同，可以把聚光灯对准特定的目标。

要把聚光灯添加到场景中，可以单击GameObject > Create Other > Spotlight 命令。此外，如果场景中已经有其他灯光，也可以把它的类型改为Spot，它便会变成聚光灯。

聚光灯只有一个尚未介绍的属性：Spot Angle。该属性确定由聚光灯发出的光的圆锥体的半径。

向场景中添加聚光灯

现在，你有机会在 Unity 中处理聚光灯了。简洁起见，这个练习使用了上一小节中创建的项目。如果你还没有完成该项目，现在必须要完成它，以便继续做这个练习。在本书配套资源中的Hour 6 下提供了这个项目的完成版本，即Hour6_SpotLight。

（1）打开前一个创建的项目。

（2）在Hierarchy视图中右键单击Point Light，并选择Rename 命令，把对象重命名为Spotlight。在Inspector中，把Type属性改为Spot。然后把灯光对象放在(0, 1, −13)处。

（3）试验聚光灯的属性。注意范围、强度和聚光角度将如何影响和改变灯光的效果。

6.1.3 定向灯光

你将在本节中处理的最后一种灯光类型是定向灯光。定向灯光类似于聚光灯，这是由于可以把它对准目标。与聚光灯不同，定向灯光会照亮整个场景。可以把定向灯光视作太阳。事实上，在前面处理地形的章节中，已经把定向灯光用作太阳了。来自定向灯光的光线将在整个场景中均匀、平行地照射。

要把定向灯光添加到场景中，可以单击GameObject > Create Other >

Directional Light命令。此外，如果在场景中已经有其他灯光，也可以把它的类型改为 Directional，它便会变成定向灯光。

定向灯光具有一个尚未介绍的属性：Cookie Size。Cookie 将在后面介绍，而这个属性控制Cookie有多大，因此也就控制了它将在整个场景中重复多少次。

向场景中添加定向灯光

我们现在将向 Unity 场景中添加定向灯光。同样，这个练习基于上一小节中创建的项目。如果还没有完成该项目，现在就要完成它，以便继续做这个练习。在本书配套资源中的Hour 6 下提供了这个项目的完成版本，即Hour6_DirectionLight。

（1）打开前一个创建的项目。

（2）在 Hierarchy 视图中右键单击 Spotlight，并选择 Rename 命令，把对象重命名为Directional Light。在Inspector中，把Type 属性改为 Directional，并把对象的旋转角度改为(75, 0, 0)。

（3）注意灯光如何照射场景中的对象。现在把灯光的位置改为(50, 50, 50)。注意灯光并没有改变。因为定向灯光是平行发光的，它的位置无关紧要，起作用的只有定向灯光的旋转角度。

（4）试验定向灯光的属性。它没有范围（范围是无限的），但是要查看颜色和强度如何影响场景。

注意：

值得一提：区域灯光

还有另外一种灯光没有在本书中介绍：区域灯光（area light）。区域灯光是Unity Pro 独有的特性，它为称为灯光贴图烘焙的过程而存在。这些主题比本书的目标更高级，基本的游戏项目不需要它们。如果想学习关于它们的更多知识，Unity具有大量的在线文档可供阅读。

6.1.4 利用对象创建灯光

由于Unity中的灯光是组件，场景中的任何对象也可以是灯光。要把灯光添加给某个对象，首先要选取该对象。然后在Inspector视图中，单击Add Component按钮，会弹出一个新列表。选择Rendering命令，然后选择Light命令。现在，你的项目就出现了一个灯光组件。给对象添加灯光的一种替代方式是选取对象，然后在菜单中单击Component > Rendering > Light命令。

给对象添加灯光时，要注意两件事：首先是对象将不会阻挡灯光，这意味着把灯光放在一个立方体内将不会阻止灯光照射；其次是给对象添加灯光不会使之发光，对象本身不会看上去像在发光一样，但它事实上在发光。

6.1.5 晕轮

晕轮是在雾气弥漫或阴云密布的情况下出现在灯光周围的发光圆环，如图 6.2 所示。之所以会出现晕轮，是因为灯光在光源四周弹射出小粒子。在 Unity 中，可以轻松地给灯光添加晕轮。每种灯光都具有一个名为Draw Halo 的复选框。如果选中它，就会为灯光绘制晕轮。

警告：

Unity的错误

从Unity 4.1起，在处理晕轮时都会存在一个错误。在编写本书时，需要执行一种解决办法，使晕轮出现在灯光周围。如果选中了Draw Halo复选框，但是灯光上什么也没有出现，就需要添加一个晕轮组件（如果出现晕轮，将不需要遵循这些步骤）。为此，可选择灯光，并单击 Component >Effects > Halo命令，现在应该会出现灯光的晕轮。此时，可以删除刚才添加的晕轮组件。对于每个场景，应该只需执行该操作一次。如果晕轮仍然没有出现，就一定要拉远摄像机镜头，因为当摄像机太近时晕轮将不会出现。

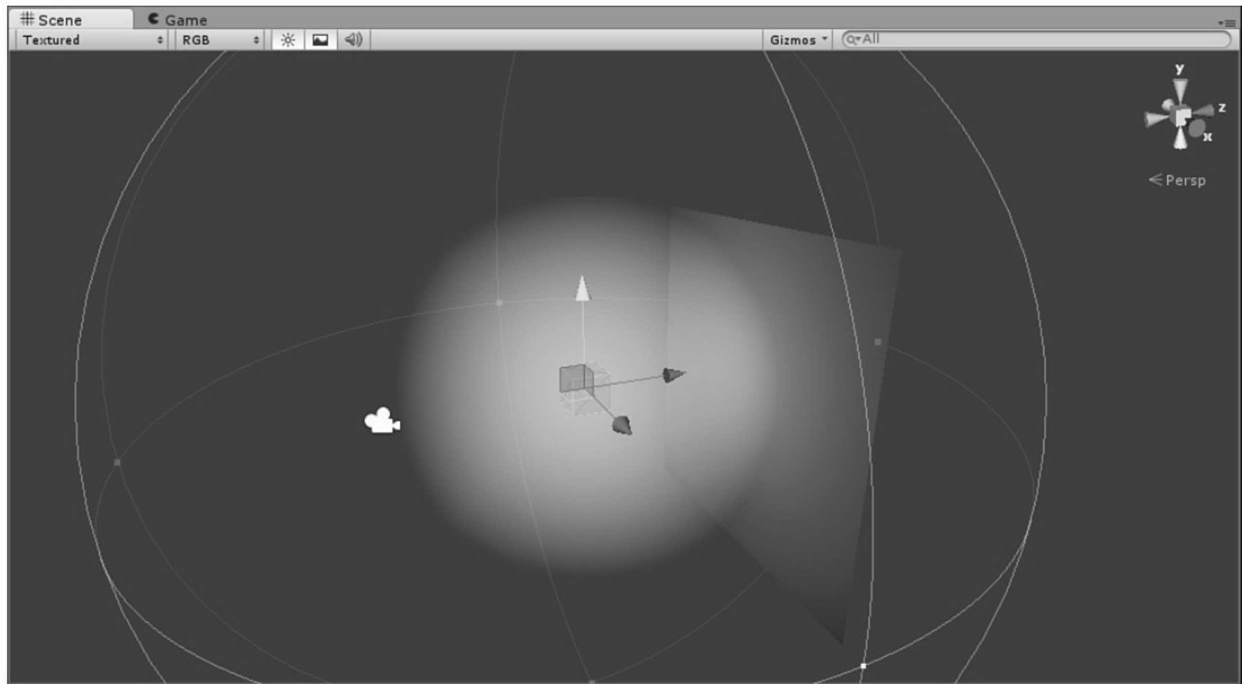


图6.2 灯光周围的晕轮

晕轮的大小是由灯光的范围确定的。范围越大，晕轮也越大。Unity 还提供了几个适用于场景中所有晕轮的属性，可以单击Edit > Render Settings命令访问这些属性。然后，渲染设置将出现在Inspector视图中，如图6.3所示。

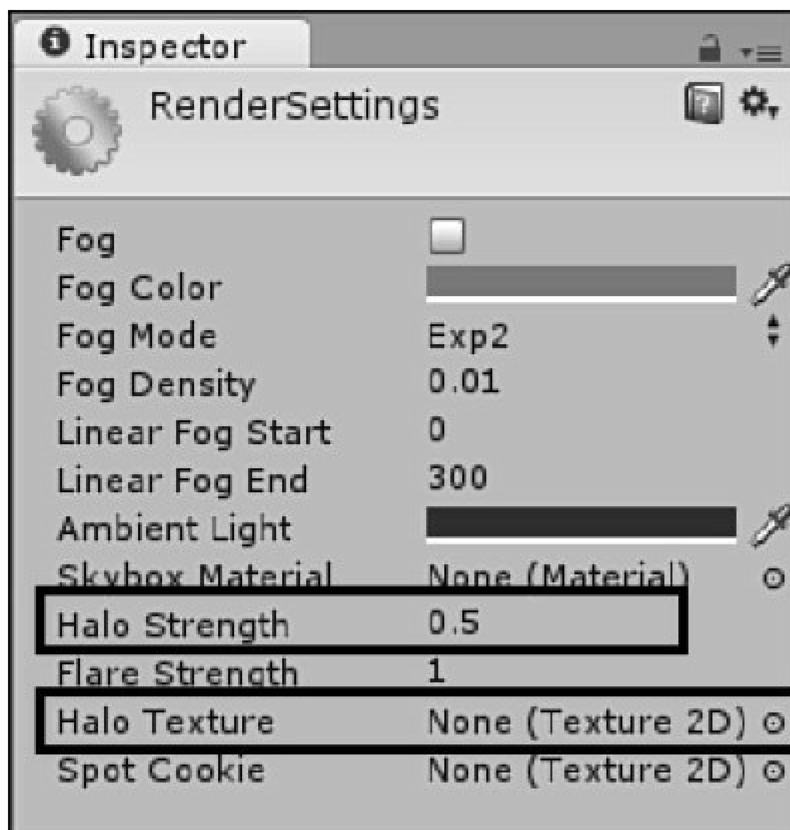


图6.3 渲染设置

Halo Strength属性确定晕轮基于灯光的范围有多大。例如，如果灯光的范围是10，并把强度设置为1，那么晕轮将全部向外扩展10个单位。如果把强度设置为0.5，那么晕轮将只向外扩展5个单位（10×0.5=5）。Halo Texture 属性通过提供一种新纹理，允许为晕轮指定一种不同的形状。如果不希望为晕轮使用自定义的纹理，可以保持它为空，并将使用默认的圆形纹理。

6.1.6 Cookie

如果你曾经在墙上点亮一盏灯，并把手放在灯光与墙之间，就可能会注意到一些灯光被手阻挡，从而在墙上留下手形阴影。在Unity中可以利用Cookie模拟这种效果。Cookie是特殊的纹理，可以把它们添加到灯光中，以表示光源是如何发光的。对于点光源、聚光灯和定向灯光，

Cookie 稍微有点不同。聚光灯和定向灯光都为 Cookie 使用黑白平面纹理。聚光灯不会重复Cookie，但是定向灯光会这样做。点光源也使用黑白纹理，但是它们必须放在立方图中。立方图是把6种纹理放在一起构成一个盒子（比如天空盒）。

给灯光添加Cookie是一个相当直观的过程，只需把纹理应用于灯光的Cookie属性即可。使Cookie工作的诀窍是：提前正确地设置纹理。要正确地设置纹理，可以在Unity中选择它，然后在Inspector窗口中更改它的属性。图6.4分别显示了用于点光源、聚光灯和定向灯光的Cookie的正确属性。



图6.4 用于点光源、聚光灯和定向灯光的Cookie的纹理属性

给聚光灯添加Cookie

让我们给聚光灯添加Cookie，以便你可以查看从开始到结束的过程。这个练习需要使用到本书配套资源中第6章的biohazard.png图像。

（1）创建一个新项目或场景。向场景中添加一个平面，把它定位于(0, 1, 0)处，并把旋转方式设置为(270, 0, 0)。

（2）选择 Main Camera，然后单击 Component > Rendering > Light 命令，并把类型更改为 Spot，给 Main Camera 添加聚光灯。把范围设置为18，把聚光角度设置为40，并把强度设置为3。

（3）把 biohazard.png 纹理从本书配套资源中拖到 Project 视图中。选取该纹理，然后在Inspector视图中把纹理类型改为Advanced。选中Alpha from Grayscale复选框，然后选中Border Mip Maps 复选框。最后，把包装模式改为Clamp，并单击Apply按钮。如果你不确信自己是否具有正确的设置，可以对照图6.4中的聚光灯设置。

（4）在选择了Main Camera 的情况下，单击并把生化危机纹理拖到灯光组件的Cookie属性中。你应该会看到生化危机标志投射到平面上，如图6.5所示。

（5）试验灯光的不同范围和强度。旋转平面，查看标志如何弯曲和扭曲。

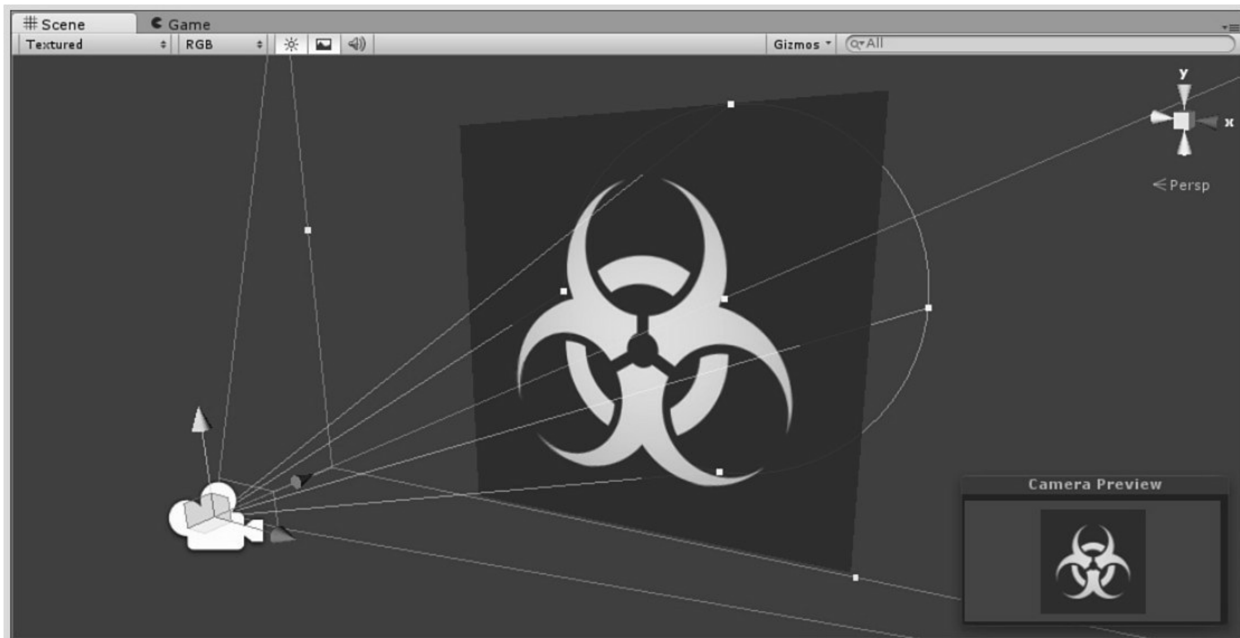


图6.5 带有Cookie的聚光灯

6.2 摄像机

摄像机是玩家观察游戏世界的视野。它提供了它们的透视图，并且可以控制事物相对于它们的状态。Unity中的所有游戏至少都有一部摄像机。事实上，无论何时创建一个新场景，总会为你添加摄像机。摄像机总是作为Main Camera 出现在层次结构中。在本节中，你将学习有关摄像机的知识，以及怎样把它们用于有趣的效果。

6.2.1 摄像机的具体分析

所有的摄像机都共享相同的属性集，这个属性集规定了它们的行为方式。表 6.2 描述了所有的摄像机属性。

表6.2 摄像机属性

属性	描述
Clear Flags	Clear Flags 属性确定在没有游戏对象的区域中摄像机将显示什么。默认是 Skybox。如果没有天空盒，摄像机默认将显示一种纯色。仅当有多部摄像机时，才应该使用 Depth Only。Don't Clear 将导致图像拖尾，只应该在编写自定义的着色器时使用
Background	Background 属性指定没有天空盒存在时的背景色
Culling Mask	Culling Mask 属性确定摄像机将会看到哪些图层。默认情况下，摄像机会看到所有的一切。不能取消选中某些图层（后面将介绍关于图层的更多知识），并且它们对于摄像机将不可见
Projection	Projection 属性确定摄像机如何查看游戏世界。两个选项是 Perspective 和 Orthographic。Perspective 摄像机感知的是 3D 中的游戏世界，其中较近的对象将较大，较远的对象将较小。如何希望游戏中具有深度，就可以使用这个设置。Orthographic 摄像机设置将忽略深度，并把所有的一切都作为平面处理

续表

属性	描述
Field of View	Field of View 属性指定摄像机可以查看多宽的区域
Clipping Planes	Clipping Planes 属性指定摄像机可以看到对象的范围。比近处的平面更近或者比远处的平面更远的对象将不会被看到
Normalized View Port Rect	Normalized View Port Rect 属性确定摄像机将投射到实际屏幕的哪个部分。默认情况下，把 x 和 y 都设置为 0，这导致摄像机从屏幕的左上角开始。把宽度和高度都设置为 1，这导致摄像机从垂直方向和水平方向 100%地覆盖屏幕。后面将更详细地介绍这个属性
Depth	Depth 属性为多部摄像机指定优先级。较低的数字将最先绘制，这意味着较高的数字可能绘制在顶部，并且实际上会隐藏它们
Rendering Path	Rendering Path 属性确定摄像机的渲染方式。应该把它保持为 Use Player Settings
Target Texture	Target Texture 属性允许为摄像机指定要绘制到的纹理，而不是屏幕。渲染纹理是一种 Unity Pro 特性
HDR	HDR（Hyper-Dynamic Range，超动态范围）属性确定 Unity 的内部灯光计算是否限制于基本的颜色范围。该属性允许高级视觉效果。目前为止，无须选中本选项

摄像机具有许多属性，但是可以设置其中大多数属性并且无须记住它们。摄像机还具有几个额外的组件：GUI Layer 允许摄像机查看GUI元素（将在本书后面介绍）；Flare Layer 允许摄像机查看灯光的镜头光晕；音频侦听器允许摄像机接收声音。如果向场景中添加更多的摄像机，将需要删除它们的音频侦听器，因为每个场景只能有一个音频侦听器。

6.2.2 多部摄像机

如果没有多部摄像机，将不可能实现现代游戏中的许多效果。幸运的是，在 Unity 场景中，可以根据需要增加摄像机。要向场景中添加新摄像机，可以单击 GameObject>CreateOther > Camera 命令。此外，还可以给已经在场景中的游戏对象添加摄像机组件。为此，可以选择对象并在Inspector 中单击Add Component，然后选择Rendering > Camera 命令添加摄像机组件。记住，给现有的对象添加摄像机组件将不会自动提供GUI Layer、Flare Layer或音频侦听器。

警告：

多个音频侦听器

如前所述，一个场景只能有一个音频侦听器。在　　Unity的老版本中，具有两个或更多的侦听器将会引发错误，并且阻止场景运行。在Unity 4中，具有多个侦听器则只会显示警告消息，尽管音频可能不会被正确地收听。在后面一章中将详细介绍这个主题。

使用多部摄像机

理解多部摄像机如何交互的最佳方式是亲自试验使用它们。这个练习重点关注的是基本的摄像机操作。

（1）创建一个新项目或场景，并添加两个立方体，把它们分别放在(−2, 1, −5)和(2, 1, 5)处。然后向场景中添加一个定向灯光。

（2）把Main Camera 移到(−3, 1, −8)处，并把它的旋转角度改为(0, 45, 0)。

（3）向场景中添加一个新摄像机（单击GameObject > CreateOther > Camera 命令），把它定位于(3, 1, −8)处，并把它的旋转角度改为(0, 315, 0)。通过取消选中组件旁边的复选框，确保禁用该摄像机的音频侦听器。

（4）运行场景。注意第二部摄像机是唯一显示的摄像机，这是由于第二部摄像机具有比Main Camera 更高的深度。先将Main Camera 移到屏幕上，然后把第二部摄像机移到其上。把Main Camera 改为1，然后再次运行场景，注意Main Camera 现在是唯一可见的摄像机。

6.2.3 拆分屏幕和图片中的图片

如你以前看到的，如果一部摄像机只是简单地绘制在另一部摄像机之上，那么在场景中具有多部摄像机并不会起到很好的作用。在本节中，你将学习使用Normalized View Port Rect属性实现拆分屏幕和图片中

的图片效果。

正常的视口（view port）实质上把屏幕作为一个简单的矩形处理。这个矩形的左上角是(0, 0)，右下角是(1, 1)。这并不意味着屏幕必须是完美的正方形。作为替代，可以把坐标视作大小的百分比。因此，坐标1表示100%，坐标0.5则表示50%。记住这一点后，把摄像机放在屏幕上就变得很容易。默认情况下，摄像机从(0, 0)处投影，并把宽度和高度都设置为1（或100%）。这导致它们将占据整个屏幕。不过，如果要更改这些数字，将获得不同的效果。

创建拆分屏幕式摄像机系统

让我们逐步创建一个拆分屏幕式摄像机系统。这种系统在双人游戏中很常见，其中玩家必须共享相同的屏幕。这个练习构建在上一小节多部摄像机的“TRY IT YOURSELF”的基础之上。

（1）打开上一小节创建的项目。

（2）确保Main Camera 所具有的深度为–1。确保将摄像机的Normalized View Port Rect属性的x和y属性都设置为0，并把w和h属性分别设置为1和0.5（100%的宽度和50%的高度）。

（3）确保第二部摄像机所具有的深度也为−1。把视口的x和y 属性设置为(0, 0.5)，这导致摄像机从屏幕中间开始向下移动。把w和h属性分别设置为1和0.5。

（4）运行场景，并且注意到两部摄像机如何同时投射到屏幕上，如图 6.6 所示。可以根据需要像这样把屏幕拆分许多次。

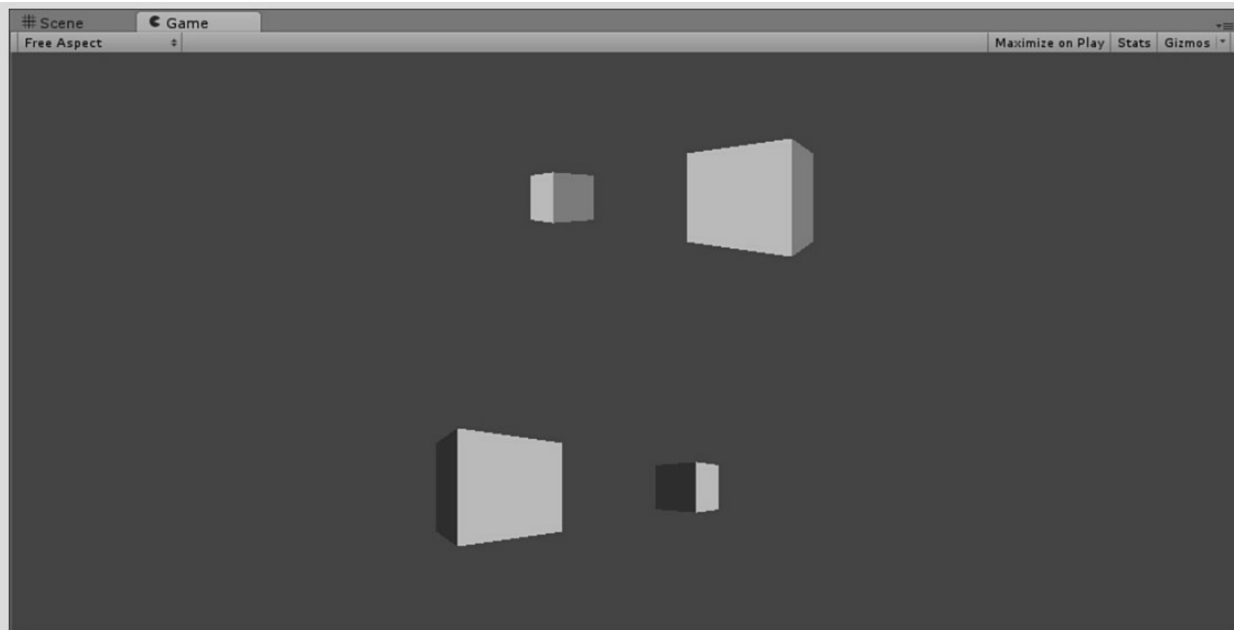


图6.6 拆分屏幕的效果

创建图片中的图片效果

图片中的图片是一种创建像迷你地图这样的效果的常见方式。利用这种效果，一部摄像机将移到特定区域中的另一部摄像机之上。这个练习构建在上一小节多部摄像机的“TRY IT YOURSELF”的基础之上。

（1）打开上一小节创建的项目。

（2）确保Main Camera 的深度为−1。确保将摄像机的Normalized View Port Rect 属性的x和y属性都设置为0，并把w和h属性都设置为1。

（3）确保第二部摄像机的深度为0。然后把视口的x和y属性设置为(0.75, 0.75)，并把w和h值都设置为0.2。

（4）运行场景。注意第二部摄像机出现在屏幕的右上角，如图 6.7所示。试验不同的视口设置，使摄像机出现在不同的角落。

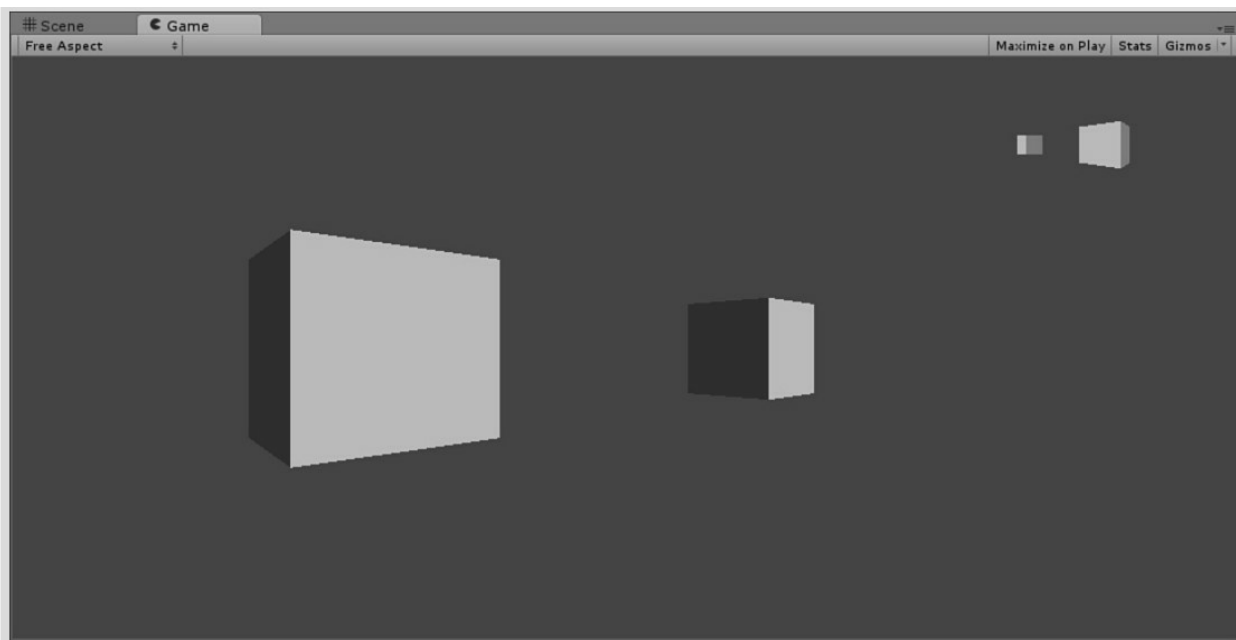


图6.7 图片中的图片效果

6.3 图层

项目和场景中具有很多对象时，通常难以组织它们。有时，你希望一些项目只能被某些摄像机看到或者只会被某些灯光照亮。有时，你希望只在某些类型的对象之间发生碰撞。Unity对这种组织方式提供的解决方案是图层。图层是类似对象的组合，使得可以以某种方式处理它们。默认情况下，有8个内置的图层，并且可以让用户定义24个图层。

警告：

图层过多

添加图层可能是无需做许多工作即可实现复杂行为的极佳方式。不过，要提出一点警告：除非必须要这么做，否则不要为项目创建图层。通常，人们在把对象添加到场景中时随意地创建图层，是考虑以后可能需要它们。这种方法可能导致组织结构的噩梦，因为你必须要记住每个图层用于什么以及它将做什么。简而言之，在需要图层时添加它们。不要仅仅因为你可以使用图层就滥用它们。

6.3.1 处理图层

每个游戏对象都开始于 Default 图层。也就是说，对象不属于特定的图层，因此它可以与其他任何对象混合在一起。在Inspector视图中可以轻松地把对象添加到图层中。在选择了对象的情况下，在Inspector视图中单击Layer下拉菜单，并选择一个新图层，所选的对象将成为它的一部分，如图6.8所示。默认情况下，有4个图层可供选择：Default、TransparentFX、Ignore Raycast 和Water。目前可以安全地忽略其中大多数选项，因为它们此时对你来说还不是非常有用。

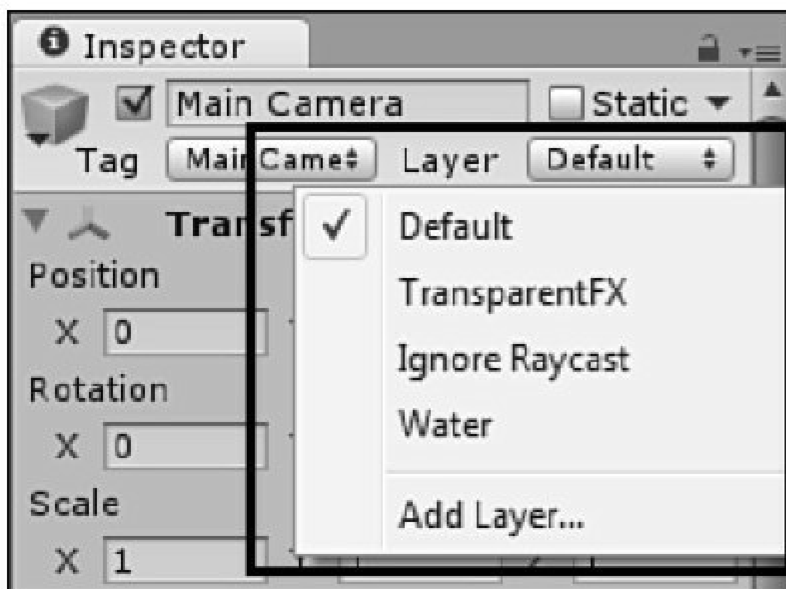


图6.8 Layer下拉菜单

尽管当前的内置图层对你不是完全有用，但是可以轻松地添加新图层。你是在 Tag Manager中添加图层的，可以用3 种方式打开Tag Manager。

选取对象，然后单击Layer 下拉菜单，并选择Add Layer 命令，如图6.8 所示。

在编辑器顶部的菜单中，单击Edit > Proje ct Settings > Tags 命令。

在场景工具栏中单击Layers 选择器，并选择Edit Layers 命令，如图6.9 所示。



图6.9 场景工具栏中的Layers选择器

一旦位于Tag Manager 中，即可在用户图层之一的右边单击，给它提供一个名称。图6.10说明了这个过程，并且显示了两个正在添加的新图层（它们是为这幅图的演示而专门添加的，你在本书配套资源中不会看见它们，除非自己手动进行添加）。



图6.10 向Tag Manager中添加新图层

6.3.2 使用图层

图层有许多应用，它们的有用性只受限于你认为可以利用它们做什么，本节将介绍3种常见的用途。

第一种是从Scene视图中隐藏图层的能力。通过在Scene视图工具栏中单击Layers选择器，如图6.9所示，可以选择哪些图层将出现在Scene视图中，以及哪些图层将不会出现。默认情况下，场景被设置成显示所

有的内容。

提示：

不可见的场景项目

Unity初学者常犯的一个错误是：意外地更改了Scene视图中可见的图层。如果你不熟悉使图层不可见的能力，这可能相当令人困惑。只需注意，每当项目应该出现在 Scene 视图中时，如果它们没有出现，就要检查Layers选择器，确保将其设置成显示所有的内容。

图层的第二种功用是使用它们排除被灯光照亮的对象。如果你是在创建自定义的用户界面、阴影系统或者使用复杂的光照系统，就会发现这个功能很有用。为了阻止图层被灯光照亮，可以选择灯光。然后，在Inspector 视图中，单击Culling Mask 属性，并取消选择你想忽略的任何图层，如图6.11所示。

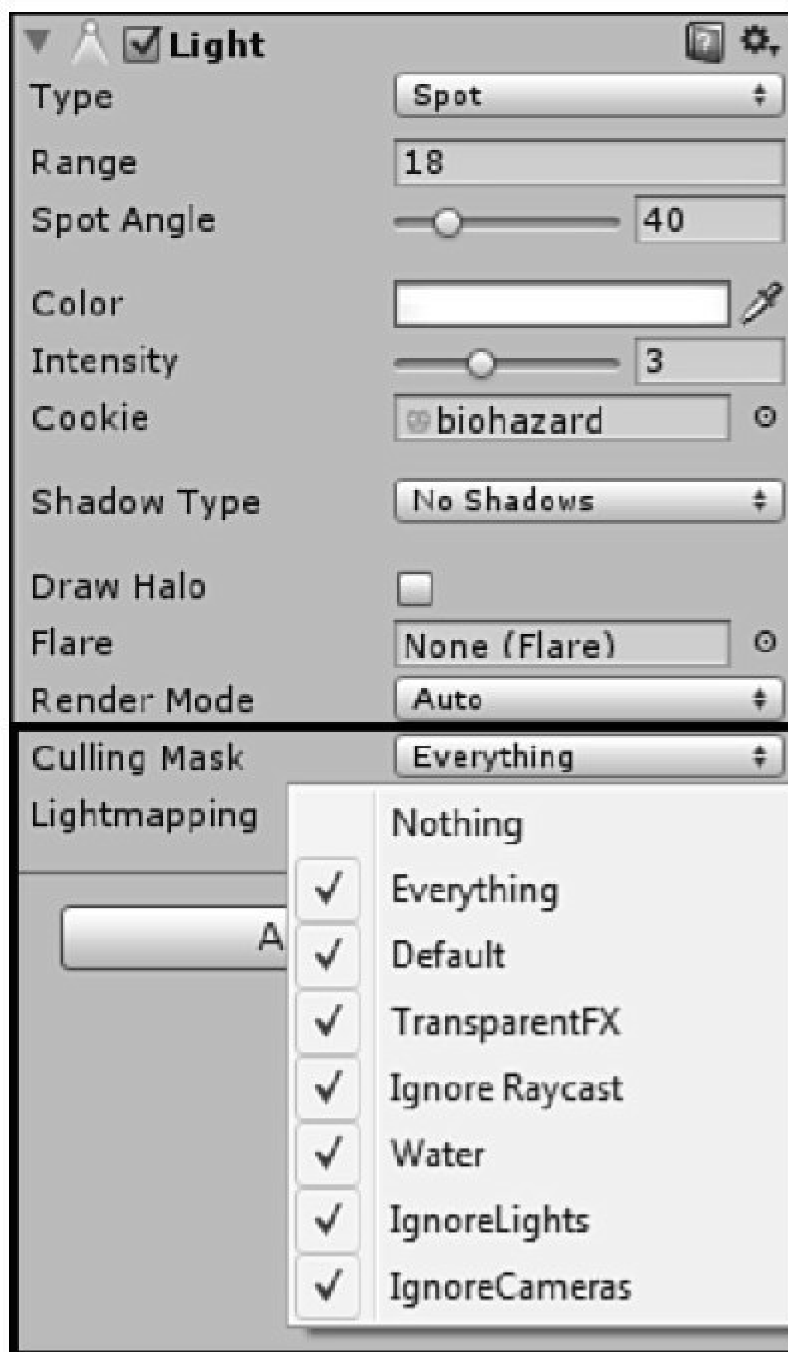


图6.11 Culling Mask属性

关于图层要知道的最后一件事是：可以使用它们确定摄像机能够看到什么，以及不能够看到什么。如果你想为单个观众使用多部摄像机构建自定义的视觉效果，这就是有用的。如前所述，要忽略图层，只需单击摄像机组件上的Culling Mask 下拉菜单，并取消选择你不希望显示的

任何图层即可。

忽略灯光和摄像机

让我们花点时间为灯光和摄像机处理图层。

（1）创建一个新的项目或场景，向场景中添加两个立方体，并把它们分别定位于(−2, 1,−5）和(2, 1, −5)处。

（2）使用前面列出的3种方法中的任意一种方法进入 Tag Manager，并添加两个新图层：IgnoreLights和IgnoreCameras，如图6.10所示。

（3）选择其中一个立方体，并把它添加到 IgnoreLights 图层中。然后选择另一个立方体，并把它添加到IgnoreCameras图层中。

（4）向场景中添加一个点光源，并把它定位于(0, 1,−7)处。在灯光的Culling Mask 属性中，取消选择 IgnoreLights 图层。注意现在只会照亮其中一个立方体，另一个立方体因为其图层的原因将被忽略。

（5）选择Main Camera，并从它的Culling Mask 属性中删除IgnoreCameras 图层。运行场景，并且注意只会显示一个没有照亮的立方体，另一个立方体被摄像机忽略了。

6.4 小结

在本章中，你学习了灯光和摄像机，并且认识了不同类型的灯光。你还学习了给场景中所具有的灯光添加Cookie和晕轮。接着，你亲自试验了摄像机，学习了有关摄像机的基础知识，以及添加多部摄像机，创建拆分屏幕和图片中的图片效果。最后，通过学习 Unity 中的图层来结束本章的内容。

6.5 问与答

问：我注意到我们略过了灯光贴图的内容，它对于学习重要吗？

答：灯光贴图是一种用于优化场景性能的有用技术。也就是说，它是一个更高级的主题，对于你在这个阶段创建的项目还不需要用到它。当你开始着手处理更高级的游戏项目时，它将变得更重要。

问：我怎样知道我想要的是透视摄像机还是正交摄像机？

答：如文中所述，一般的经验法则是：对于3D游戏和效果，需要透视摄像机；对于2D游戏和效果，则需要正交摄像机。

6.6 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

6.6.1 问题

1．如果你想要利用一种灯光照亮整个场景，那么应该使用哪种类型的灯光？

2．可以向场景中添加多少部摄像机？

3．你可以创建多少个用户定义的图层？

4．什么属性确定了哪些图层将被灯光和摄像机忽略？

6.6.2 答案

1．定向灯光是可以均匀地应用于整个场景的唯一一种灯光。

2．你可以具有所需要的多部摄像机。

3．24个。

4．Culling Mask 属性。

6.6.3 练习

在这个练习中，你将有机会操作多部摄像机和多种灯光。在构造这个练习时可以具有一定的灵活性，因此可以自由地发挥你的创意。

1．创建一个新的场景或项目，向场景中添加一个球体，并把它置于(0, 0, 0)处。

2．向场景中添加4个点光源，并把它们分别放置在(−4, 0, 0)、(4, 0, 0)、(0, 0, −4)和(0,0, 4)处，并给它们分别提供自己的颜色。设置范围和

强度，在球体上创建想要的视觉效果。

3．从场景中删除Main Camera（右键单击Main Camera，并选择Delete）。向场景中添加4 部摄像机，并对其中3部摄像机禁用音频侦听器。然后把它们分别定位于(2, 0, 0)、(–2, 0, 0)、(0, 0, 2)和(0, 0, −2)处，并围绕y 轴旋转它们，直到它们都面对球体为止。

4．更改4部摄像机上的视口设置，使得你利用全部4部摄像机实现了一种拆分屏幕式效果。应该在屏幕的每个角落显示一部摄像机，它们都占据屏幕大小的1/4，如图6.12所示。这一步留给你完成。如果你不知所措，在Hour 6 资源中提供了这个练习的完成版本，它的名称是Hour6_Exercise。

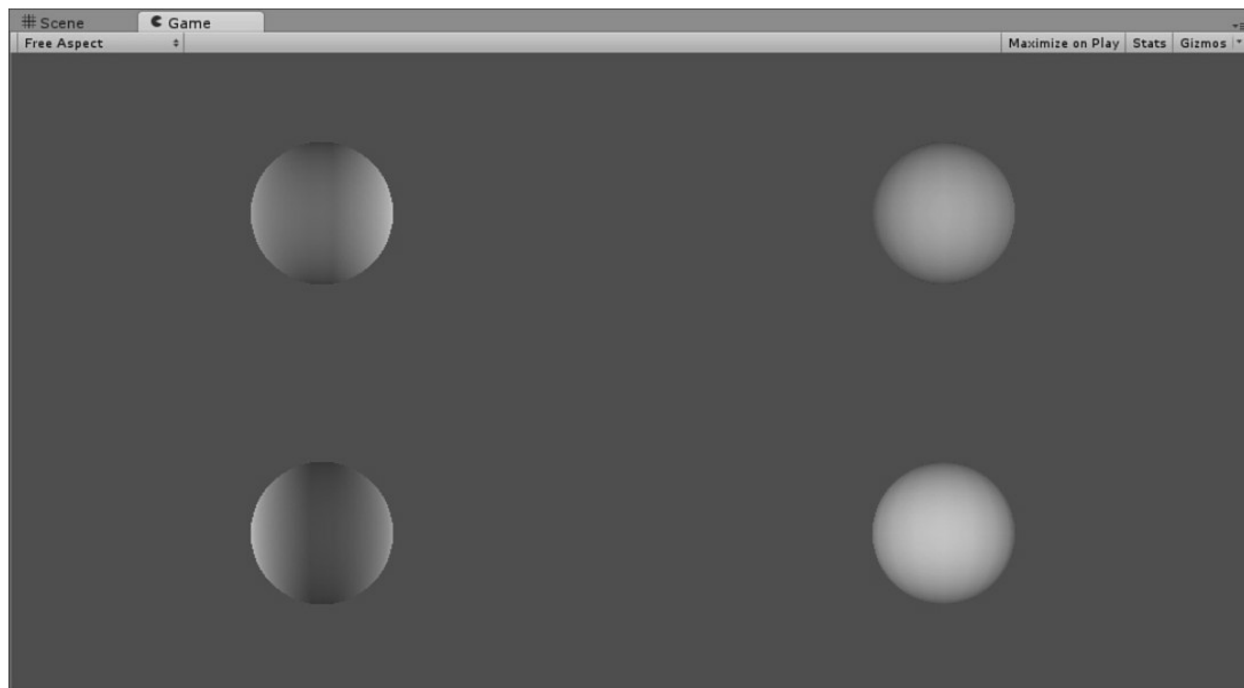


图6.12 完成的练习

第7章 第1款游戏：Amazing Racer

在本章中你将学到：

怎样设计基本的游戏；

怎样应用地形知识构建特定于游戏的世界；

怎样向游戏中添加对象以提供交互性；

怎样测试和调整完成的游戏。

在本章中，你将消化迄今为止所学的知识，并使用它们构建你的第一个 Unity 游戏。你首先将了解游戏的基本设计元素。接着，将构建游戏发生的世界。然后，将添加一些交互性对象，以使玩家能够玩游戏。最后，将开始玩游戏，并执行任何必要的调整以改进体验。

提示：

完成的项目

一定要遵循本章中的指导来构建完整的游戏项目。如果你不知所措，可以在本书配套资源中的Hour 7下找到该游戏的完成版本。如果需要帮助或灵感，可以看一看它。

7.1 设计

游戏开发的设计部分用于提前规划游戏的所有主要特性和组件。可以把它视作是布置一份蓝图，使得实际的构造过程要顺畅得多。在创建游戏时，有许多时间通常都花在完成设计上。由于你将在本章中创建的游戏相当基本，因此设计阶段将完成得很快。在创建这款游戏时，你需要重点关注规划的3个领域：理念、规则和需求。

7.1.1 理念

这款游戏背后的思想很简单。从一个区域的一边开始，并迅速行进到另一边。在前进的道路上将会出现山丘、树木和各种障碍物。你的目标是看看可以多快地完成它。之所以为你的第一款游戏选择这种游戏理念，是因为它突出显示了你迄今为止处理过的所有领域。此外，由于你还没有学习在Unity中编写脚本，因此不能添加非常精巧的交互。将来的游戏将更复杂。

7.1.2 规则

每个游戏都必须具有一组规则。规则服务于两个目的：第一，它们告诉你玩家实际上将怎样玩游戏；第二，因为软件是一个许可的过程（参见下面的“注意：许可的过程”），规则指定了可供玩家征服挑战所采用的动作。用于Amazing Racer 游戏的规则如下。

没有获胜或失败的条件，只有完成的条件。当玩家进入完成区域时，就完成了游戏。

玩家总是从相同的地点复活，完成区域也总是位于相同的地点。

将会出现水障。无论何时玩家陷入水障中，都会把该玩家移回复活点（spawn point）。

游戏的目标是尝试尽可能获得最快的时间。这是一条隐含的规则，并且没有明确地构建到游戏中。作为替代，将在游戏中构建一些线索，暗示玩家这就是目标。其思想是：玩家将基于提供给他们的信号凭直觉期望更快的时间。

注意：

许可的过程

在创建游戏时，始终要记住软件是一个许可的过程。这意味着除非明确指定允许什么，否则它将对玩家不可用。例如，如果玩家希望爬树，但是你没有创建任何方式让玩家爬树，那么就不允许执行那个动作。如果你没有给玩家跳跃的能力，他们就不能跳跃。你希望玩家能够做的一切事情都必须构建到游戏中。记住：不能假定任何动作，必须为一切都做好规划！

注意：

术语

本章中使用了下面一些新术语。

复活：复活是一个过程，玩家或实体通过它进入游戏。

复活点：复活点是玩家或实体复活的位置，游戏中可以有一个或多个复活点，它们可以是静止的，或者是四处移动的。

条件：条件是一种触发器形式。获胜条件是使玩家赢得游戏的事件（比如积累足够的点数）；失败条件是使玩家输掉游戏的事件（比如丢失所有的单击点数）。

游戏控制器：游戏控制器规定了游戏的规则和流程。它负责知道何时赢得或输掉游戏（或者只是游戏结束了）。可以把任何对象指定为游戏控制器，只要它总是在场景中即可。通常，把一个空对象或者 Main Camera 指定为游戏控制器。

7.1.3 需求

设计过程中的另一个重要步骤是确定游戏将需要哪些资源。一般来讲，游戏开发团队由多人组成。其中一些人将从事设计，其他人则负责编写程序或创建艺术。团队的每位成员在开发过程中的每个步骤当中都需要做某件事情来提高效率。如果每一个人都等待某件事情完成才能开始工作，那么将会出现许多开始和停止。作为替代，你可以提前确定资源，以便在事物需要之前就可以创建它们。下面列出了Amazing Racer游戏的所有需求。

一块矩形区域的地形。地形需要足够大，以展示一场具有挑战性的比赛。地形中应该具有一些障碍，以及指定的复活点和终点，如图7.1所示。

用于地形的纹理和环境效果。它们是在Unity标准资源中提供的。

一个复活点对象、一个完成区域对象和一个水障对象。将在Unity中生成它们。

一个角色控制器。这是由Unity标准资源提供的。

一个图形用户界面（GUI）。将在本书配套资源中为你提供它。

一个游戏控制器。将在Unity中创建它。

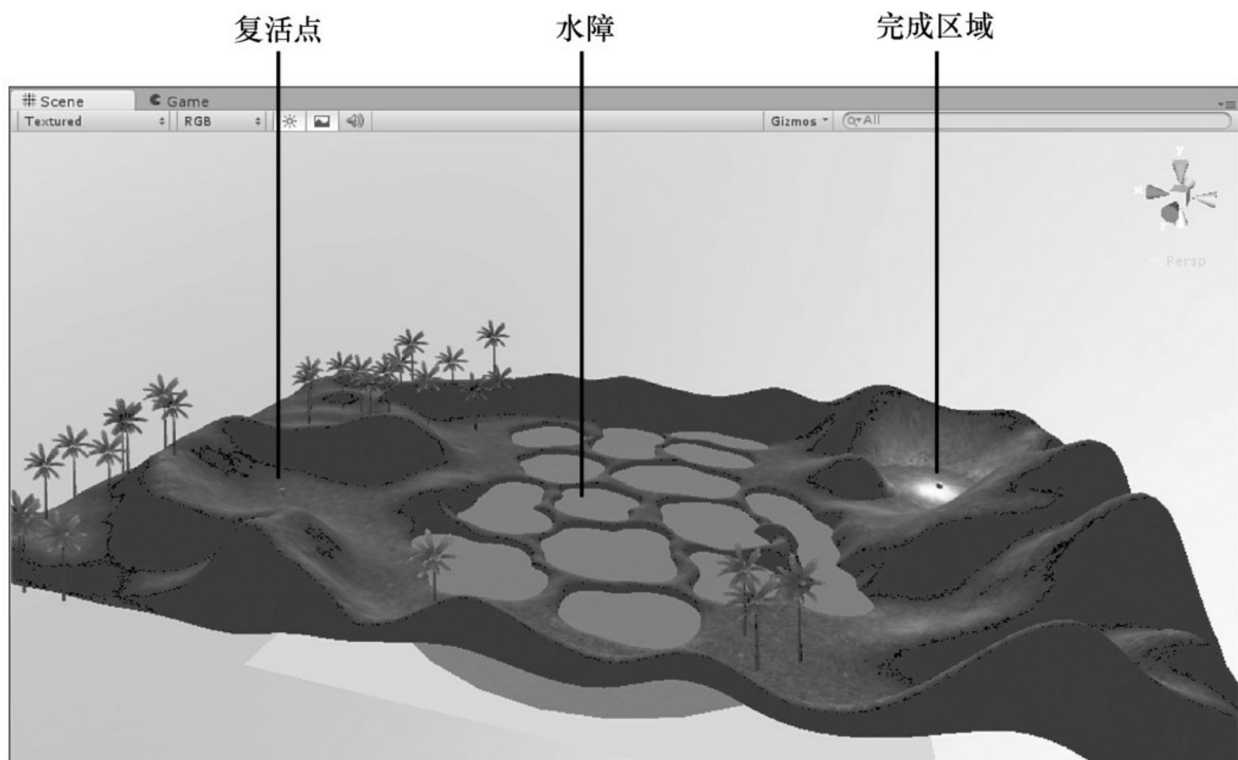


图7.1 AmazingRacer 游戏的一般地形布局

7.2 创建游戏世界

既然你已经在纸上写下了游戏的基本思想，现在就可以开始构建它。可以在许多位置开始构建一款游戏。对于这个项目，可以从游戏世界开始。由于这是一款线性赛车游戏，游戏世界的长度将大于它的宽度（或者它的宽度将大于它的长度，这依赖于你如何查看它）。将使用许多Unity标准资源快速创建该游戏。

7.2.1 雕刻游戏世界

可以用许多方式创建这个地形。每一个人都会有不同构思。为了简化这个过程，将为你提供一幅高度图。它用于确保每一个人在本章中都将具有相同的经历。要雕刻地形，可以遵循下面这些步骤。

（1）在名为Amazing Racer的文件夹中创建一个新项目，并向该项目中添加一个地形。

（2）把该地形的分辨率设置为 200（宽）×100（长）×100（高）（在 Terrain Settings 的Resolution区域中设置它）。

（3）在用于第7章（Hour 7）的本书配套资源中定位文件terrain.raw。导入该文件作为地形的高度图（通过在Terrain Settings的Heightmap 区域中单击Import Raw命令）。

（4）在资源下面创建一个Scenes文件夹，并把当前场景另存为Main。

现在应该雕刻地形，以匹配本书中的游戏世界。可以自由地执行微小的调整和修改，以使其具有你喜欢的样子。

警告：

构建你自己的地形

在本章中，你将基于所提供的高度图构建一款游戏。高度图已经为你准备好了，以便你可以迅速体验游戏开发的过程。不过，你也可以选择构建自定义的游戏世界，以使这款游戏独一无二，并且是属于你自己的一款游戏。不过，如果你这样做，就要当心提供给你的一些坐标和旋转角度可能不匹配。如果你想构建自己的游戏世界，就要注意打算放置对象的位置，并把它们相应地放在游戏世界中。

7.2.2 添加环境

此时，你可以开始纹理化地形，并向其中添加一些环境效果。你需要导入以下程序包（单击Assets > Import Package 命令）。

Terrain Assets

Skyboxes

Water

你现在可以有一点自由，按你所喜欢的方式装饰游戏世界。下面的建议是指导原则，可以以自己的喜好来安排事情。

向场景中添加一种定向灯光，旋转它以适应你的偏好。

纹理化地形。示例项目使用以下纹理：Grass (Hill)用于平坦部分，Cliff (Layered Rock)用于陡峭部分，Grass&Rock用于它们之间的区域。

向场景中添加一个天空盒（单击Edit > Render Settings 命令）。示例项目为它的天空盒使用Sunny1 Skybox。

向地形中添加一些树木。应该稀疏地放置树木，并且主要放在平坦的表面上。

向场景中添加一些基本的水（在 Project 视图中从 Assets\Standard Assets\Water (Basic)文件夹中拖动 Daylight Simple Water）。把水放在(88, 29, 49)处，并把它缩放成(50, 1, 50)。

现在应该就准备好了地形，并且准备在它上面行走了。一定要花相当多的时间进行纹理化，以确保具有良好的混合和逼真的外观。在示例项目中还有一些额外的事物没有展示出来，但是你可能想添加它们，包括以下几点。

雾。

水障周围的青草。这可能会把它们遮掩一点，并且增加了难度。

用于定向灯光的光晕，用于模拟太阳。需要旋转定向光源，以匹配天空盒的太阳图像（如果具有天空盒的话）。

7.2.3 角色控制器

在开发的这个阶段，你将希望向场景中添加一个角色控制器。

（1）单击Assets > Import Package > Character Controller命令，导入标准的角色控制器。

（2）从Assets\Character Controllers 文件夹中把First Person 控制器资源拖入场景中。

（3）把First Person 控制器（它将被命名为Player 并设置为蓝色）放置在(160, 32, 64)处。在y轴上把控制器旋转260°，使之面朝正确的方向。

一旦角色控制器位于场景中并且定位了它，就可以播放场景。一定要四处看看，寻找需要修补或平滑的任何区域。要注意边界。寻找你能够逃离游戏世界的任何区域，将需要抬高这些位置，使得玩家将不能脱离地图。在这个阶段，一般将修正地形的任何基本的问题。

提示：

脱离游戏世界

一般来讲，游戏关卡将具有一些水或者其他一些障碍，以阻止玩家退出已开发的区域。如果游戏利用重力，玩家可能会脱离游戏世界的边

缘。你总是希望创建某种方式，阻止玩家到达某个他们不应到达的区域。这个游戏项目使用高高的护堤使玩家保持在游戏区域内。在用于第7章（Hour　7）的本书配套资源中提供的高度图有意给出了几个位置，玩家可以从中爬出。看看你是否能够找到并校正它们。

7.3 游戏化

现在，你拥有了一个可以玩游戏的游戏世界。你可以四处逛逛，并在一定程度上体验这个游戏世界。现在，所遗漏的部分正是游戏本身。目前，你所拥有的只是一个玩具，只可以玩一玩。你想要的是一款游戏，可它只是一个具有一定规则和一定目标的玩具。把某件东西转变成游戏的过程称为游戏化（gamification），这就是本节将要做的事情。如果你遵循了前面的步骤，那么游戏项目现在看上去应该如图 7.2 所示。下面几个步骤是添加一些游戏控制对象以便进行交互，为那些对象应用游戏脚本，并把它们彼此联系起来。

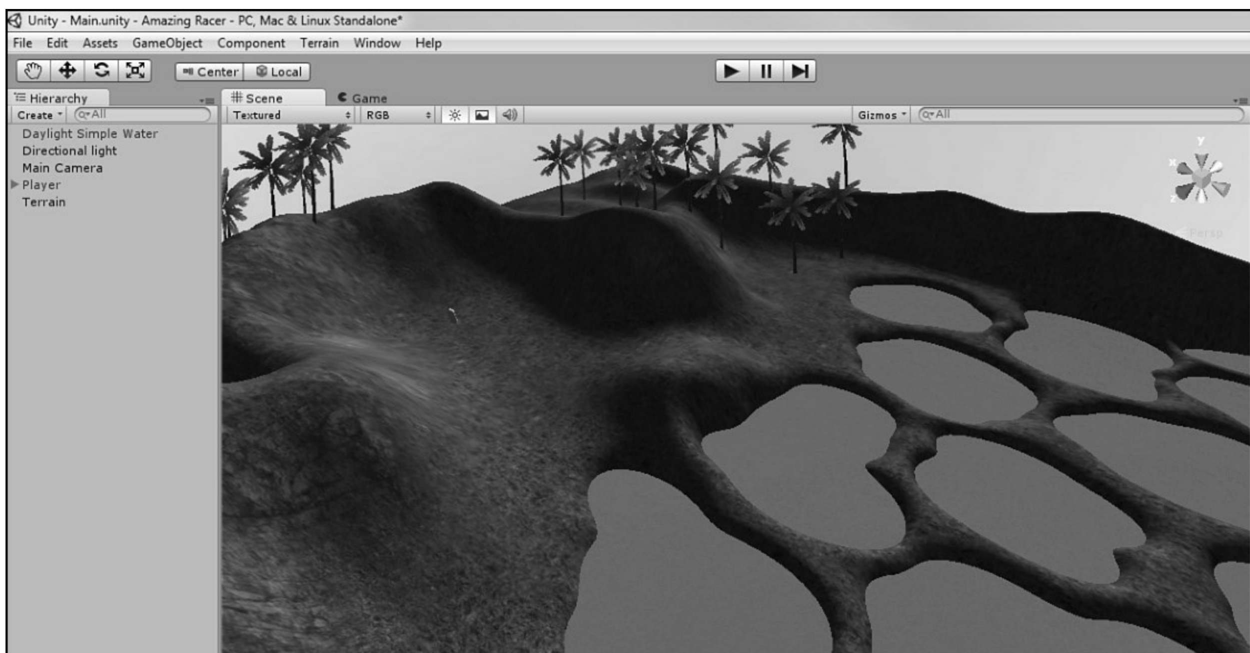


图7.2 Amazing Racer游戏的当前状态

注意：

脚本

脚本是为游戏对象定义行为的代码段。你还没有学习在Unity中编写脚本。不过，要创建交互式游戏，脚本是必须的。考虑到这一点，我

们为你提供了制作这款游戏所需的脚本。我们努力使脚本尽可能小，以便你可以理解这个项目的大部分内容。可以自由地在文本编辑器中打开脚本，并阅读它们所做的事情。在第8章和第9章中将更详细地介绍脚本。

7.3.1 添加游戏控制对象

如在7.1.3节中所定义的，需要4个特定的游戏控制对象。第一个对象是复活点。这将是一个简单的游戏对象，只用于告诉游戏在哪里复活玩家。要创建复活点，可以遵循下面这些步骤。

（1）向场景中添加一个空的游戏对象（单击GameObject > Create Empty命令），并把它定位于(160, 32, 64)处。

（2）在Hierarchy视图中把空对象重命名为SpawnPoint。

接下来，创建水障检测器。它应该是一个位于水下的简单平面，并具有一个触发碰撞器（将在本书后面更详细地介绍），用于检测玩家何时落入水中。要创建检测器，可以遵循下面这些步骤。

（1）向场景中添加一个平面（单击GameObject > Create Other > Plane 命令），把它定位于(86, 27, 51)处，并把该平面缩放为(10, 1, 10)。

（2）在Hierarchy视图中把平面重命名为WaterHazardDetector。

（3）在Inspector视图中选中Mesh Collider 组件上的Is Trigger 复选框，如图7.3 所示。

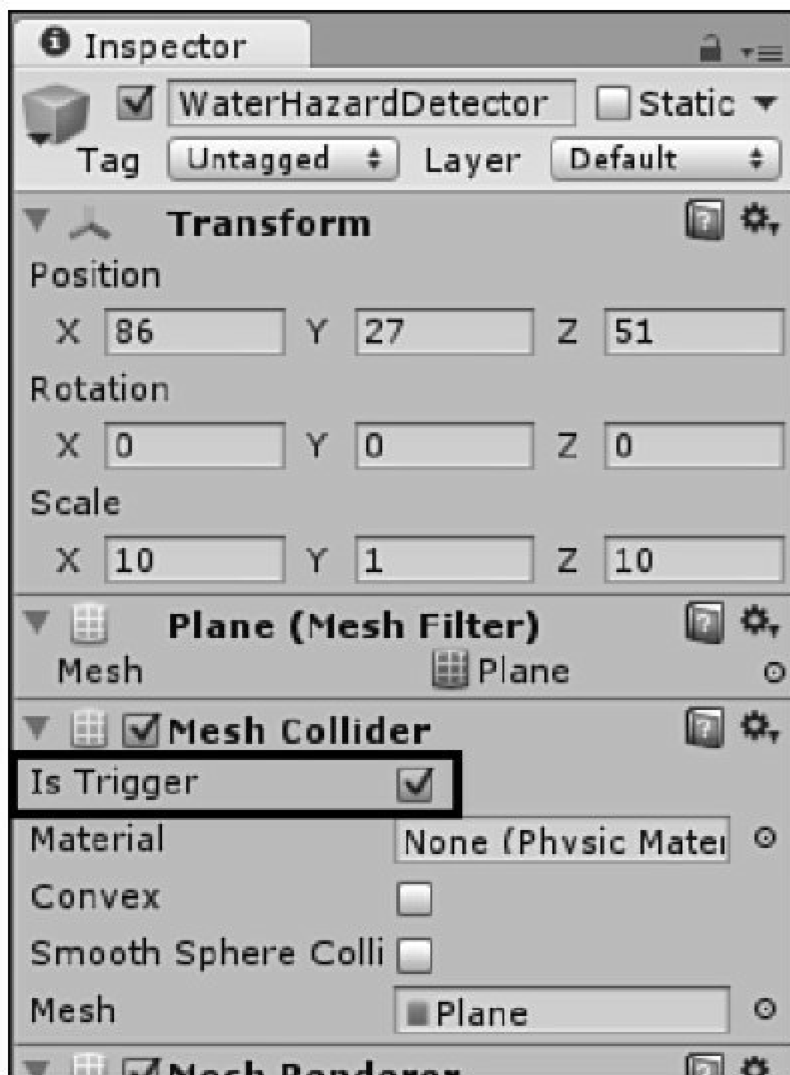


图7.3 WaterHazard Detector对象的Inspector视图

接下来，你希望给游戏添加完成区域。这个区域将是一个简单的对象，它上面带有一个点光源，使得玩家知道要去往哪里。该对象连接有一个胶囊碰撞器（capsule collider），以便它了解玩家何时可以进入该区域。要添加完成区域对象，可以遵循下面这些步骤。

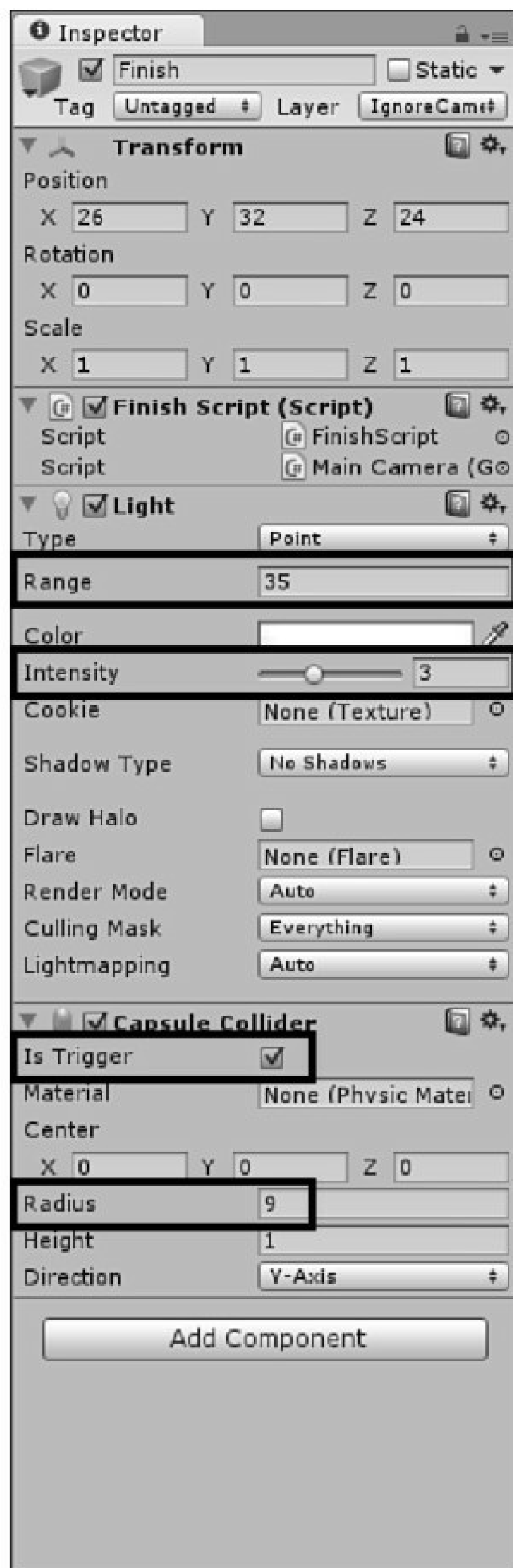
（1）向场景中添中一个空的游戏对象，并把它定位于(26, 32, 24)处。

（2）在Hierarchy视图中把该对象重命名为Finish。

（3）给完成区域对象添加一个灯光组件（选取该对象，单击

Component > Rendering > Light命令）。如果它的类型还不是Point，就把类型更改为Point，并把范围和强度分别设置为35和3。

（4）选取完成区域对象并单击Component > Physics > Capsule Collider命令，给该对象添加一个胶囊碰撞器。在Inspector视图中把Radius属性改为9，并选中IsTrigger复选框，如图7.4所示。



Inspector
Finish
Static
Tag
Untagged
Layer
IgnoreCam
Transform
Position
X 26
Y 32
Z 24
Rotation
X 0
Y 0
Z 0
Scale
X 1
Y 1
Z 1
Finish Script (Script)
Script
FinishScript
Script
Main Camera (G
Light
Type
Point
Range
35
Color
Intensity
3
Cookie
None (Texture)
Shadow Type
No Shadows
Draw Halo
Flare
None (Flare)
Render Mode
Auto
Culling Mask
Everything
Lightmapping
Auto
Capsule Collider
Is Trigger
Material
None (Physic Mate
Center
X 0
Y 0
Z 0
Radius
9
Height
1
Direction
Y-Axis
Add Component

图7.4 Finish对象的Inspector视图

需要创建的最后一个对象是游戏控制对象。这个对象从技术上讲不需要存在，可以代之以只把它的属性应用于游戏世界里的其他一些持久的对象，比如 Main Camera。不过，你一般都会创建它自己的对象，以防止意外的删除。在开发的这个阶段，游戏控制对象是非常基本的，以后将更多地使用它。要创建游戏控制对象，可以遵循下面这些步骤。

（1）向场景中添加一个空的游戏对象。

（2）在Hierarchy视图中把该游戏对象重命名为GameControl。

7.3.2 添加脚本

如前所述，脚本指定了游戏对象的行为。在本节中，将对游戏对象应用脚本。此时，理解这些脚本用于做什么对你来说并不重要。你需要做的第一件事是把脚本添加到项目中。

（1）在Project视图中的Assets下面创建一个Scripts文件夹。

（2）在用于第7章（Hour　7）的本书配套资源中找到Scripts文件夹。

（3）从本书配套资源的Scripts文件夹中单击并把脚本拖到Unity中的Scripts文件夹中。应该有3个脚本：FinishScript、GameControlScript和RespawnScript。

一旦脚本位于项目中，应用它们就很容易。要应用一个脚本，只需把它从Project视图中拖到你想应用它的任何对象上即可，如图7.5所示。应用下面这些脚本。

对Finish 游戏对象应用FinishScript。

对GameControl 对象应用GameControlScript。

对WaterHazardDetector对象应用RespawnScript。

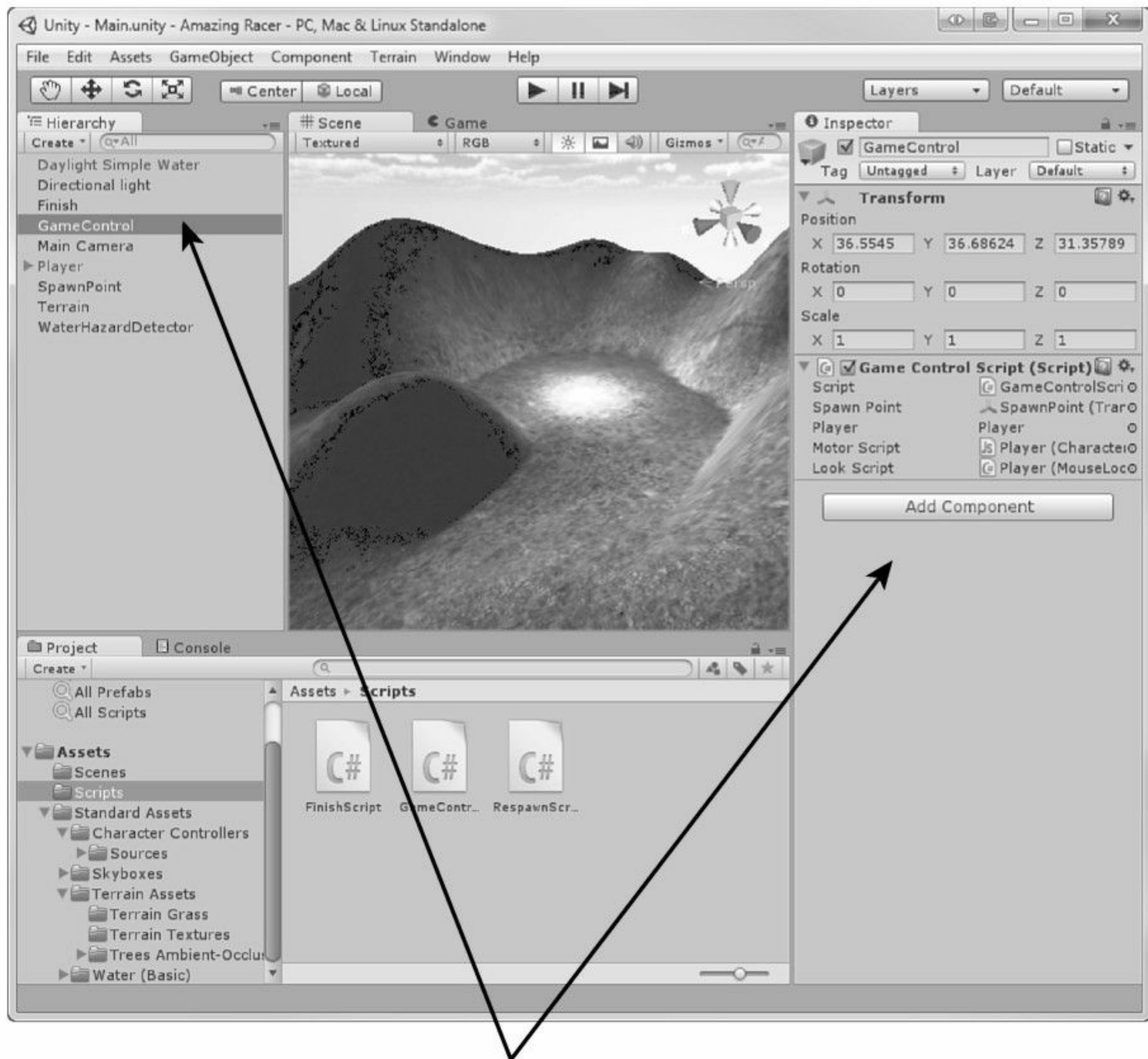


图7.5 通过把脚本拖到游戏对象上来应用它们

7.3.3 把脚本连接在一起

如果通读脚本，就会注意到它们都具有用于其他对象的占位符。这些占位符允许一个脚本与另一个脚本交流。你会看到，对于脚本中存在的每个占位符，在Inspector视图中用于那个脚本的组件中都会有一个属性。就像脚本一样，通过单击和拖动把对象应用于占位符，如图7.6所

示。

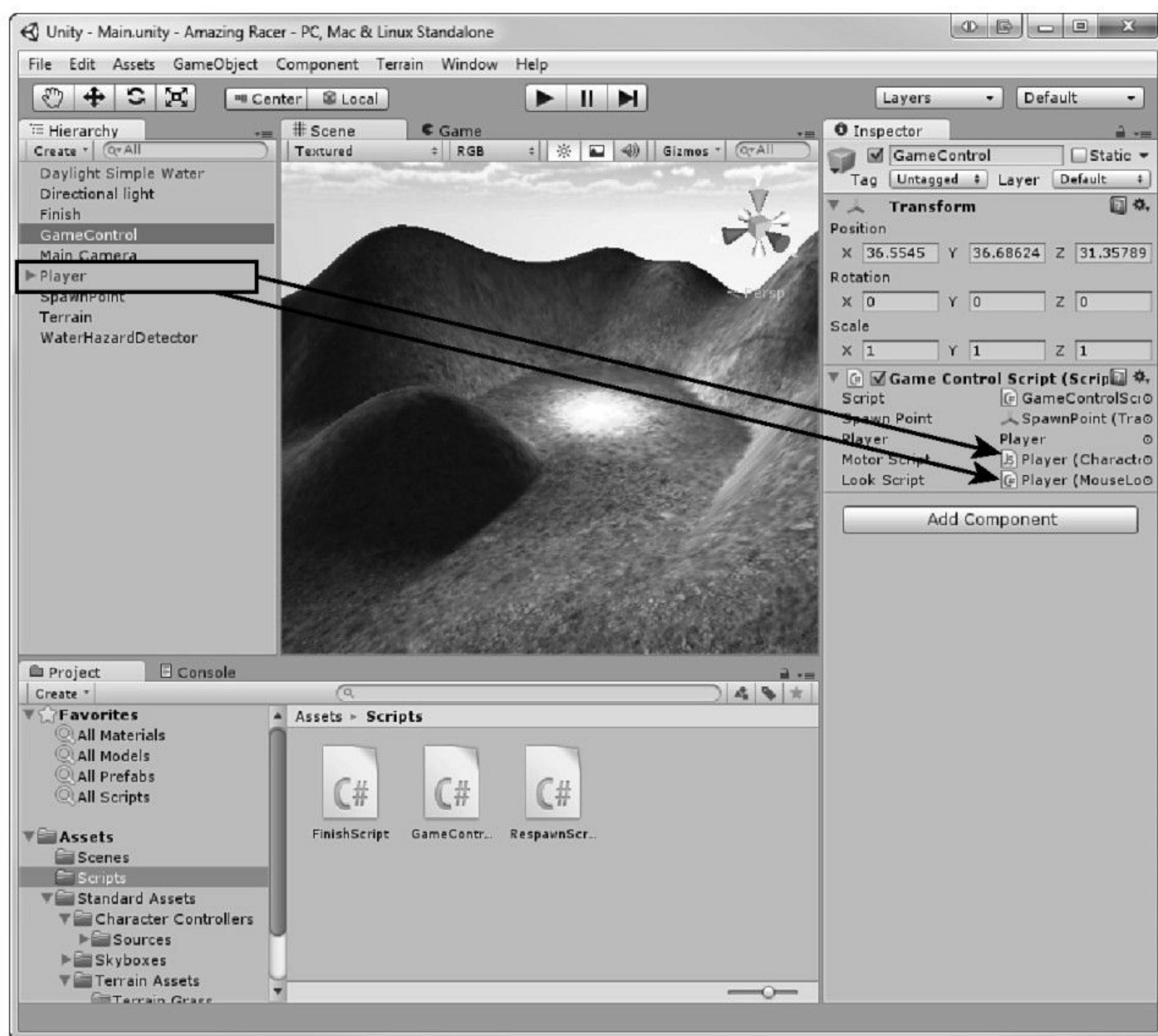


图7.6 把游戏对象移到占位符上

首先，开始把对象与 WaterHazardDetector 连接起来。在 Hierarchy 视图中选取WaterHazardDetector，并且注意到它怎样具有Respawn Script 组件。这是在上一节中应用复活脚本的结果。你还注意到复活组件具有一个 Respawn Point 属性，这个属性是用于在前面创建的SpawnPoint游戏对象的占位符。在选取了WaterHazardDetector对象之后，从Hierarchy 视图中单击并把SpawnPoint 对象拖到Respawn Script 组件的Respawn Point 属性之上。现在，无论何时玩家陷入水障中，都将把他们移回关

卡开始位置的复活点处。

要设置的下一个对象是Finish游戏对象。选取Finish游戏对象，从Hierarchy视图中单击并把GameControl 对象拖到Inspector 视图中的Finish Script 组件的Game Control Script属性上。现在，无论何时玩家进入完成区域，游戏控制都将会得到通知。

需要设置的最后一个对象是 GameControl。要正确地设置这个控制，可以遵循下面这些步骤。

（1）单击并把 SpawnPoint 对象拖到 GameControl 的 Game Control Script 组件的 Spawn Point属性上。

（2）单击并把Player 对象（这是角色控制器）拖到GameControl 的Game Control Script的Player 属性、Motor Script 属性和Look Script 属性上。

（3）需要在 Player 角色控制器上禁用摄像机的鼠标状态脚本。为此，可以在 Hierarchy视图中展开Player对象（单击Player左边的箭头以展开它），然后选择嵌套在Player下面的Main Camera。在Inspector视图中定位Mouse Look (Script)组件，并取消选中它，如图7.7所示。

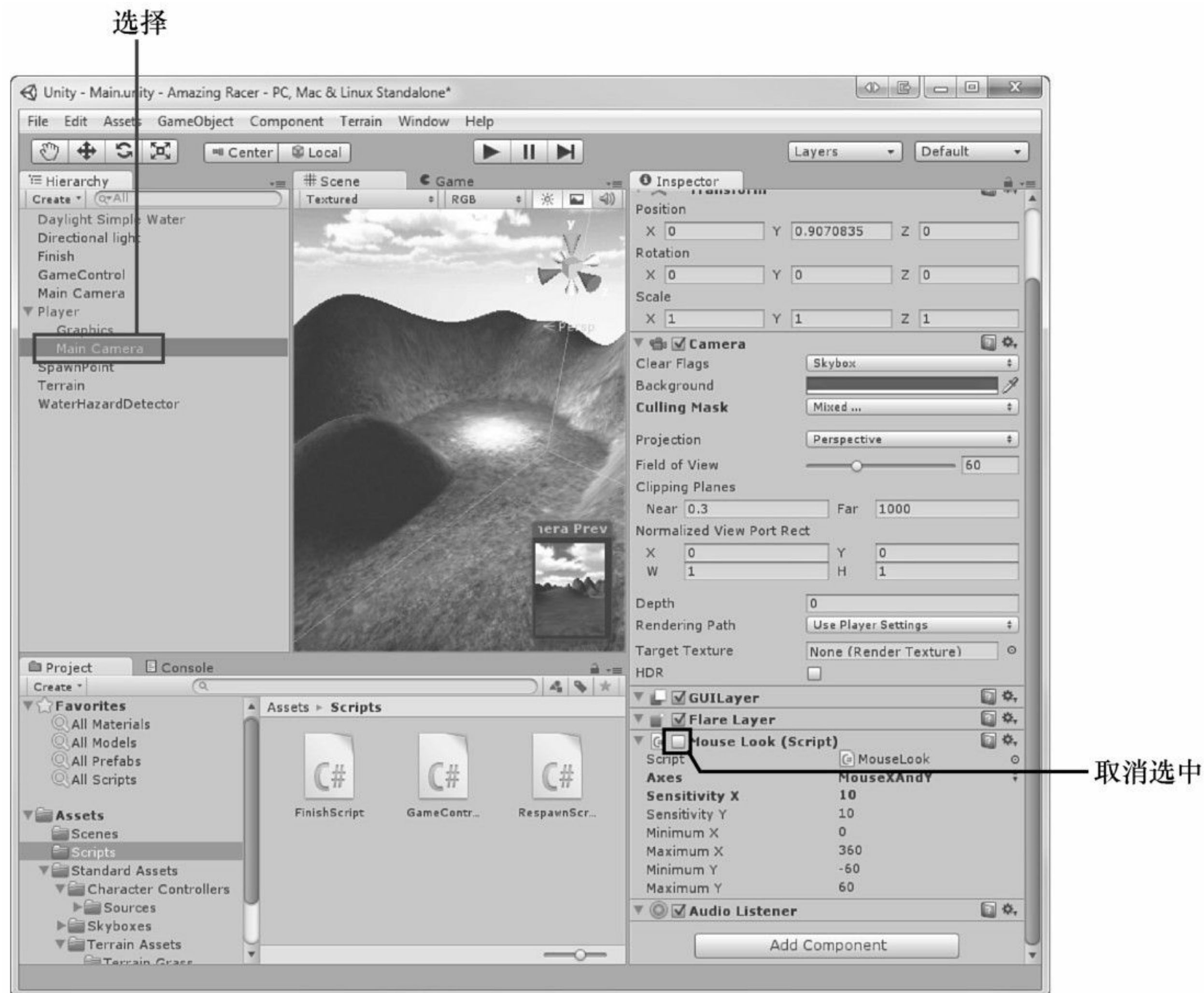


图7.7 在玩家的摄像机上取消选中 Mouse Look组件

这样就把游戏对象都连接好了。游戏现在完全具有可玩性！游戏中的一些部分目前可能还没有意义，但是随着你对它了解得越多，它将变得越直观。

7.4 游戏测试

你的游戏现在就完成了，但是还不能就此止步。现在，你必须开始游戏测试的过程。游戏测试是指带着找出错误或者不如你所愿的事情的意图来玩游戏。在很多时候让其他人测试你的游戏可能是有益的，以便他们可以告诉你什么对于他们是有意义的，以及他们发现什么是令人愉悦的。

如果你遵循前面描述的所有步骤，应该不会找出任何错误（通常称为bug）。不过，确定哪些部分是有趣的过程完全取决于游戏制作者的意见。因此，这个部分留给你完成。可以玩一玩游戏，并看看你不喜欢什么。要重点注意那些不让你感到愉悦的事情。不过，不要只关注负面的东西，还要发现你所喜欢的事情。你更改这些事情的能力可能目前会受到限制，因此要把它们记下来。如果给你机会，就要规划如何把游戏改变得更好。

目前可以进行调整以使游戏变得更令人愉悦的一件简单的事情是玩家的速度。如果玩过两次这款游戏，就可能会注意到角色移动得太慢，并且这可能使人感到游戏很长并且拖拖拉拉。为了使角色变得快一些，需要修改 Player 对象上的 Character Motor (Script)组件。在Inspector视图中展开Movement属性，并且修改最大前进速度，如图7.8所示。示例项目把它设置为12。试试这个设置，看看你是否喜欢它。试试更快或更慢的速度，并选择一个你喜欢的速度。

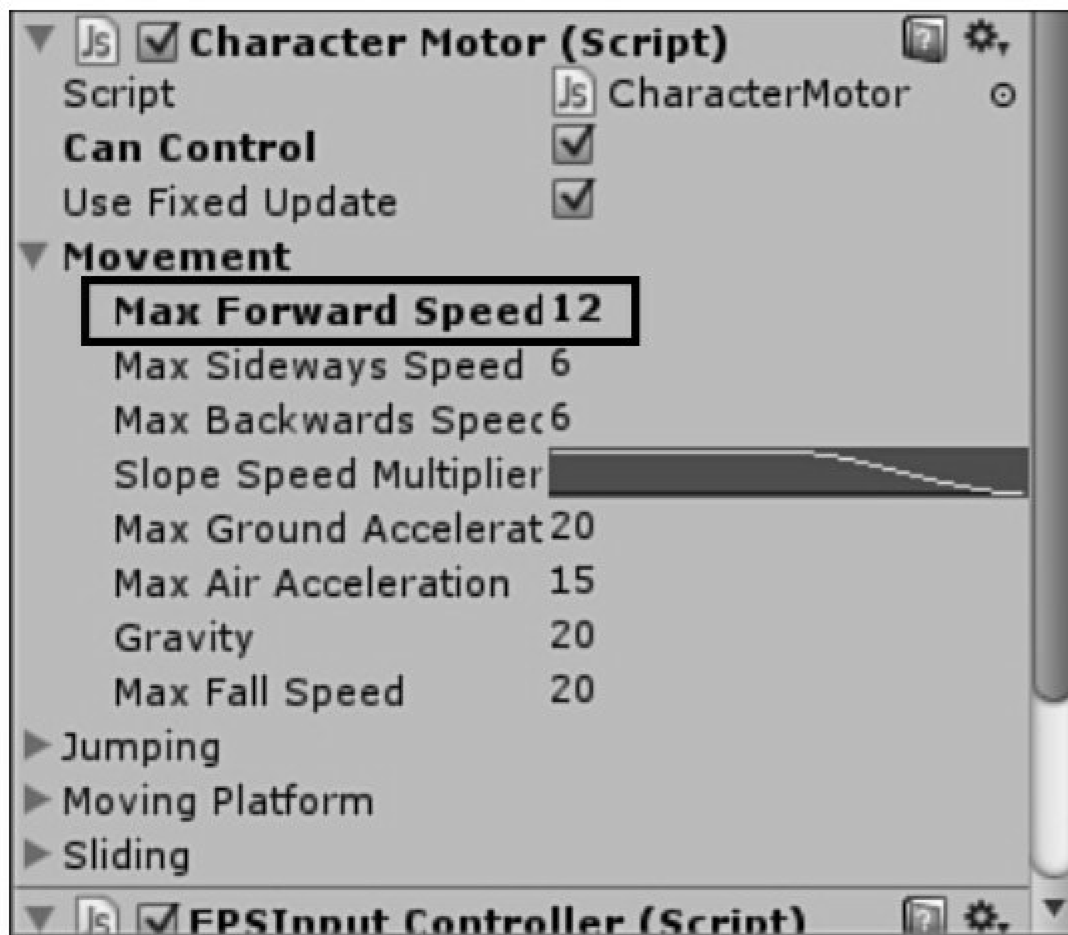

Character Motor (Script)
Script
CharacterMotor
Can Control
Use Fixed Update
Movement
Max Forward Speed 12
Max Sideways Speed 6
Max Backwards Speec 6
Slope Speed Multiplier
Max Ground Accelerat 20
Max Air Acceleration 15
Gravity 20
Max Fall Speed 20
Jumping
Moving Platform
Sliding
FPSInput Controller (Script)

图7.8 更改玩家的速度

7.5 小结

在本章中，你在 Unity 中制作了自己的第一款游戏。你首先设计了游戏理念、规则和需求的多个方面。接着，你构建了游戏世界，并且添加了环境效果。然后，你添加了交互性所需的游戏对象。你对那些游戏对象应用了脚本，并把它们联系在一起。最后，你对游戏进行了测试，并且注明了你喜欢和不喜欢的事情。

7.6 问与答

问：这似乎超越了我的理解力，我做错了什么事情吗？

答：根本没有！这个过程对于不习惯它的人可能感到非常生疏。保持阅读和学习书中的材料，慢慢就会开始熟悉这个过程。你能够做的最好的事情是注意对象如何通过脚本彼此产生联系。

问：你没有介绍如何构建和部署游戏。为什么？

答：构建和部署游戏在本书后面有专门一章加以介绍。在构建游戏时有许多事情要考虑，此时，你只应该把注意力集中在开发它所需的理念上。

问：如果没有脚本，为什么我们不能制作游戏？

答：如前所述，脚本定义了对象的行为。如果没有某种形式的交互式行为，将很难具有一款条理分明的游戏。在第8章和第9章学习编写脚本之前，就在第7章中构建一款游戏的唯一原因是：应该在转向不同的内容之前强化你已经学过的主题。

7.7 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

7.7.1 问题

1．什么是游戏的需求？

2．什么是这款游戏的获胜条件？

3．在这种地形中，建议使用多少种纹理以获得一种自然混合的外观？

4．哪个对象负责控制游戏的流程？

5．为什么我们要测试游戏？

7.7.2 答案

1．需求是制作游戏需要创建的资源列表。

2．这是一个有意捉弄人的问题！这款游戏没有明确的获胜条件。当玩家这一次到达的时间比上一次更短时，就认为他或她获胜了。不过，没有以任何方式把它构建到游戏中。

3．3种：青草、青草和岩石、岩石。

4．游戏控制器。在这款游戏中，它被命名为GameControl。

5．要发现错误，以及确定游戏的哪些部分像我们希望的那样工作。

7.7.3 练习

对于制作游戏，最令人愉快的体验便是可以使它们像你所想的那样

运行。遵循指导可能是一种良好的学习经历，但是将不能获得制作一款自定义游戏的满足感。在这个练习中，你将有机会稍微修改一下游戏，以使事物更独特。至于你想如何修改游戏，则完全取决于你自己。下面列出了一些建议。

尝试添加多个完成区域。看看你在放置它们之后，是否可以给玩家提供更多的选择。

修改地形，使之具有更多的或不同的障碍。只要像水障那样构建这些障碍（包括脚本），它们就会工作得很好。

尝试具有多个复活位置，并且使其中一些障碍把你移到第二个或第三个复活点。

修改天空和纹理，创建一个陌生的游戏世界。并且提供独特的游戏世界体验。

第8章 编写第1部分的脚本

在本章中你将学到：

Unity中的脚本的基础知识；

如何使用变量；

如何使用运算符；

如何使用条件；

如何使用循环。

迄今为止，你学习了如何在Unity中制作对象。不过，那些对象有一点令人厌烦。放在那里的立方体有多少用？给立方体提供某个自定义的动作，使之显得更有趣一些，这样效果会好得多。你所需要的是脚本。脚本是用于为对象定义复杂的或非标准行为的代码文件。在本章中，你将学习编写脚本的基础知识。你首先将考虑在Unity中如何开始使用脚本，学习如何创建脚本以及使用脚本编程环境。然后，你将学习脚本语言的多种成分，这些成分包括变量、运算符、条件语句和循环。

提示：

示例脚本

在用于第8章（Hour 8）的本书配套资源中，有本章提到的多个脚本和编码结构。一定要检查它们，以进行额外的学习。

警告：

编程新手

如果你以前从未编写过程序，这似乎令人感到奇怪和混淆。在学习本章时，要竭尽全力关注事物是如何构造的，以及为什么要那样构造它

们。记住：程序设计完全是合乎逻辑的。如果程序没有做你希望它做的事情，这是因为你没有告诉它如何正确地去做。有时，需要你改变自己的思维方式。要慢慢消化吸收本章的知识，并且一定要进行实践。

8.1 脚本

如前所述，脚本是一种定义行为的方式。它们像其他组件一样附加在Unity中的对象上，并给它们提供交互性。在Unity中，使用脚本一般包括3个步骤。

（1）创建脚本。

（2）把脚本附加到一个或多个游戏对象上。

（3）如果脚本需要它，就用值或其他游戏对象填充属性（这一步以后再讨论）。

8.1.1 创建脚本

在创建脚本前，最好在Project视图中的Assets文件夹下面创建一个Scripts文件夹。一旦具有一个包含所有脚本的文件夹，就只需右键单击该文件夹，并选择Create > C# Script 命令。一旦创建了脚本，在继续执行下面的操作前，需要给它提供一个名称。

注意：

脚本语言

Unity允许用C#、JavaScript或Boo编写脚本。本书使用C#语言编写所有的脚本。注意：并没有哪一种语言一定优于另一种语言。如果你偏爱某种语言，可以自由地使用该语言工作。

一旦创建了脚本，就可以查看和修改它。在 Project 视图中单击脚本，将使你能够在Inspector视图中查看脚本的内容，如图8.1所示。在Project视图中双击脚本将打开默认的编辑器，这将使你能够向脚本中添加代码。假定你安装了默认的组件并且没有更改任何内容，那么双击一

个文件将打开MonoDevelop开发软件，如图8.2所示。



图8.1 脚本的Inspector视图预览

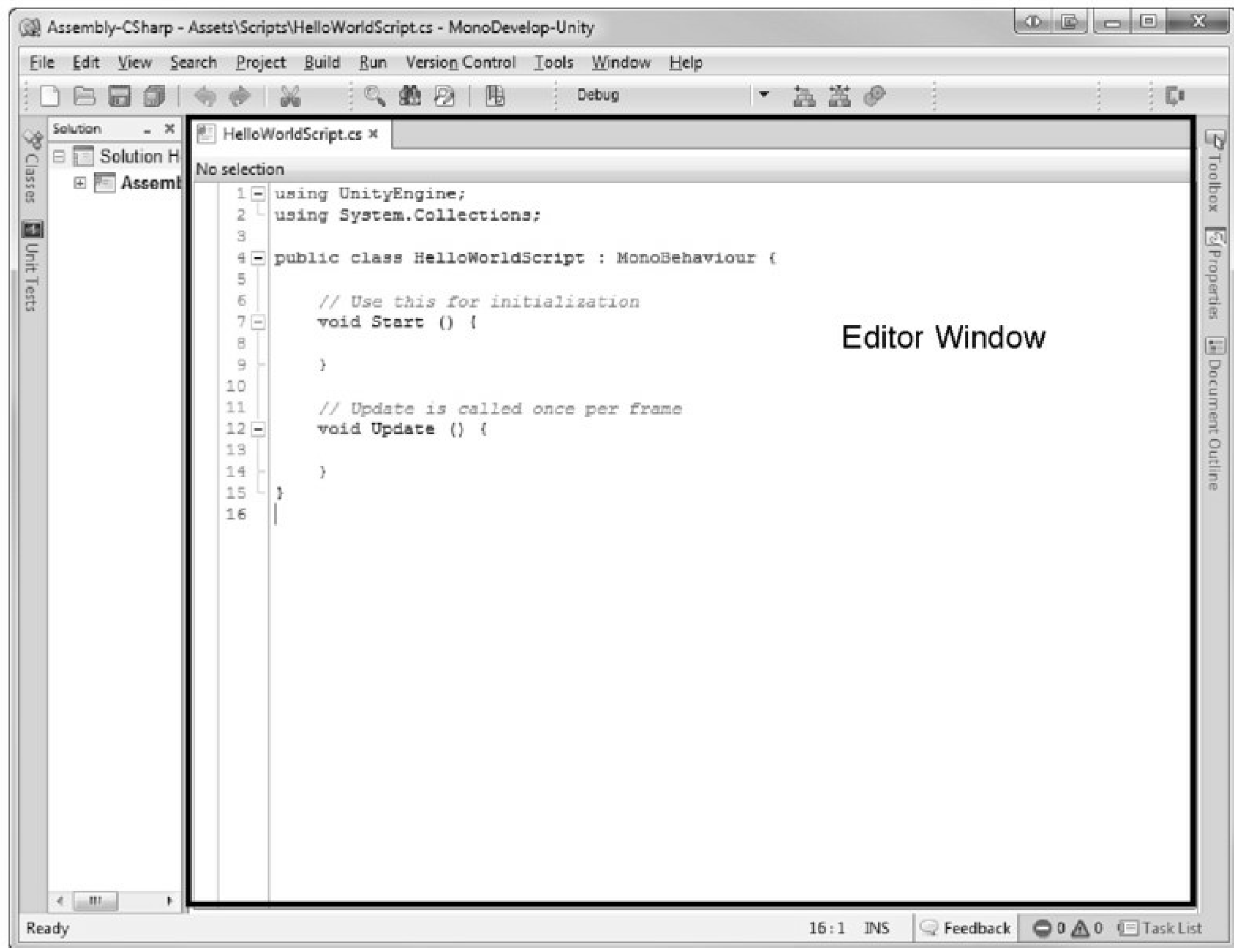


图8.2 高亮显示了编辑器窗口的Mono Develop软件

创建脚本

让我们创建一个在本节中使用的脚本。

（1）创建一个新的项目或场景，并向Project视图中添加一个Scripts文件夹。

（2）右键单击 Scripts 文件夹，并选择 Create > C# Script 命令，然后把脚本命名为HelloWorldScript。

（3）双击新的脚本文件，并等待MonoDevelop打开。在MonoDevelop的编辑器窗口中，如图8.2所示，清除所有的文本，并利用下面这个程序清单中的代码替换它们：

```
using UnityEngine;
```

```
using System.Collections;
public class HelloWorldScript : MonoBehaviour {
    // Use this for initialization
    void Start (){
        print ("Hello World");
    }
    // Update is called once per frame
    void Update (){
    }
}
```

（4）单击File>　　　　Save命令或者按下Ctrl+S组合键（Mac上的Command+S组合键），保存脚本。回到Unity中，在Inspector视图中确认脚本已被更改，并且运行场景。注意没有任何事情发生。创建了这个脚本，但是直到把它附加到对象上之后，它才会工作。接下来将介绍这一点。

注意：

MonoDevelop

MonoDevelop是一份与Unity一起免费提供的健壮、复杂的软件。它实际上不是Unity的一部分。因此，我们不会深入介绍它。对于MonoDevelop，你目前需要熟悉的唯一部分是编辑器窗口。如果你还需要知道MonoDevelop的其他任何方面，本章会在需要的地方介绍它。

8.1.2 附加脚本

要把脚本附加到游戏对象上，只需在Project视图中单击脚本，并把它拖到对象上即可，如图8.3所示。可以在Hierarchy视图、Scene视图或Inspector视图中把脚本拖到对象上（假定已经选取了对象）。一旦把脚

本附加到对象上，它将变成该对象的一个组件，并且在Inspector视图中是可见的。

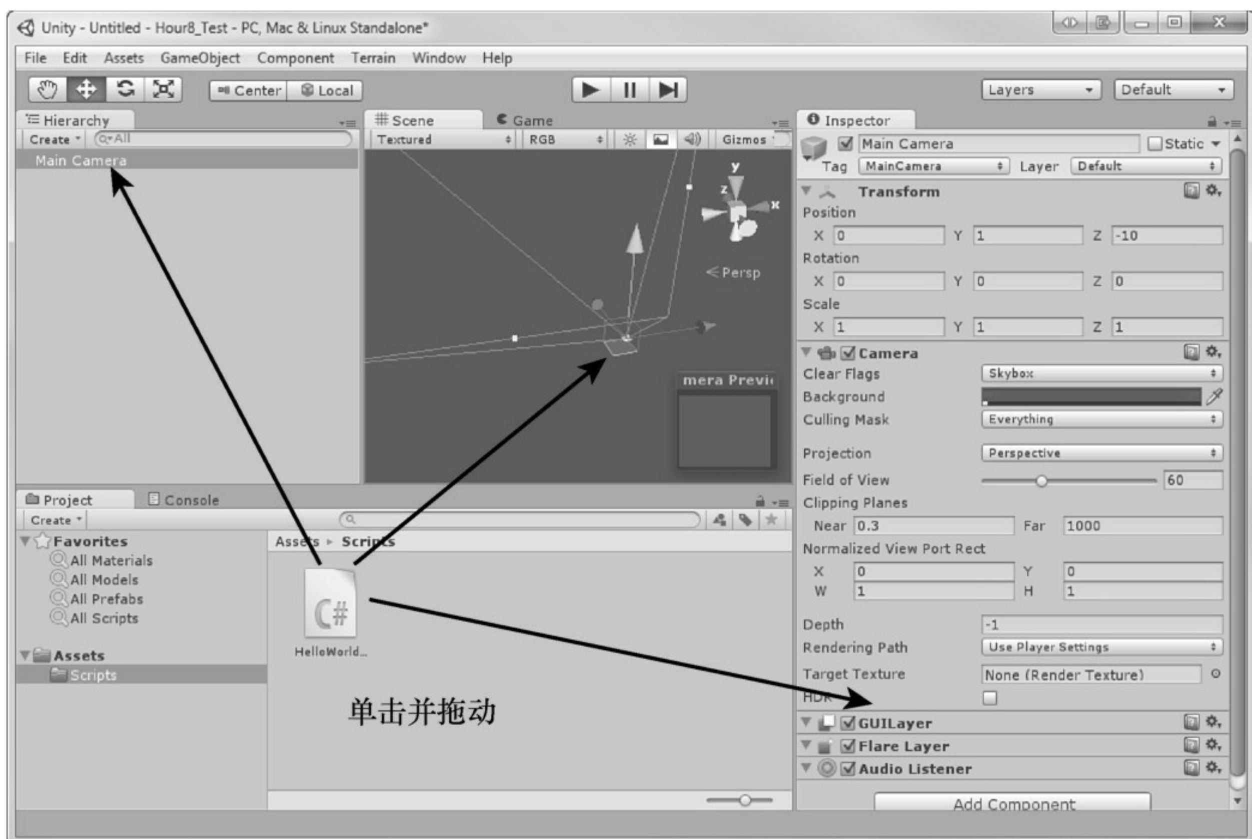


图8.3 单击并把脚本拖到想要的对象上

要查看它的实际操作，可以把前面创建的 HelloWorldScript 附加到 Main Camera 上。你现在应该会在Inspector视图中看到一个名为Hello World Script (Script)的组件。如果运行场景，将会看到“Hello World”出现在屏幕底部，如图8.4 所示。

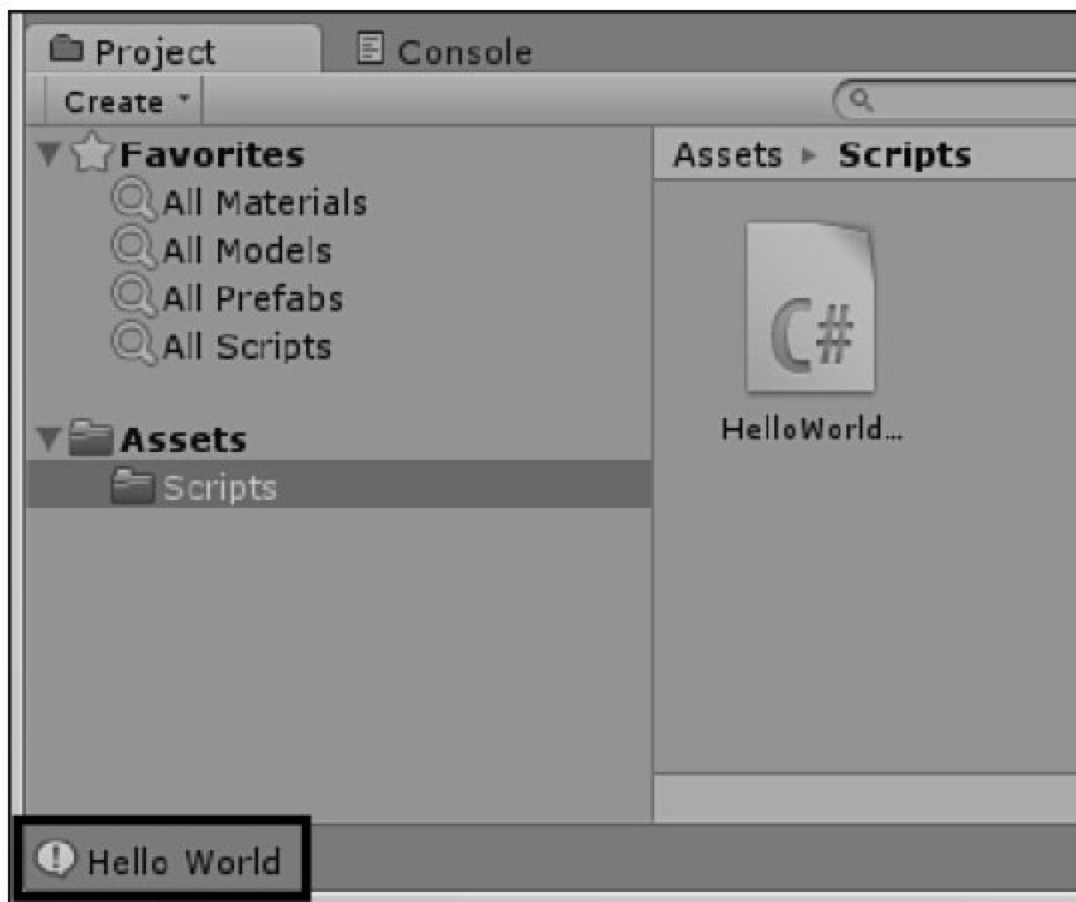


图8.4 运行场景时输出单词“Hello World”

8.1.3 一个基本脚本的详细分析

在上一节中，你修改了一个脚本，以把一些文本输出到屏幕上，但是没有解释脚本的内容。在本节中，你将查看适用于每个新的 C#脚本的默认模板。注意：用 JavaScript 或Boo编写的脚本将具有相同的组件，即使它们看上去稍有不同。程序清单8.1包含在创建新脚本时 Unity 为你生成的全部代码，这个程序清单假定创建的脚本文件被命名为HelloWorldScript。

程序清单8.1 默认的脚本代码

```
using UnityEngine;
using System.Collections;
```

```
public class HelloWorldScript : MonoBehaviour {
    // Use this for initialization
    void Start () {
    }
    // Update is called once per frame
    void Update () {
    }
}
```

可以把这段代码分解为3个部分。

1．using部分

第一部分列出了这个脚本将使用的库。它看起来如下：

```
using UnityEngine;
using System.Collections;
```

一般来讲，你无需改变这个部分，并且在目前，应该使之保持不变。

2．类声明部分

下一个部分被称为类声明。每个脚本都包含一个以脚本命名的类，它看起来如下所示：

```
public class HelloWorldScript : MonoBehaviour { }
```

开始大括号与封闭大括号之间的所有代码都将是这个类的一部分，因此也是脚本的一部分。所有的代码都应该出现在这些大括号之间。同样，如上，你无需改变它，并且目前应该使之保持不变。

3．类内容

类的开始和封闭大括号之间的部分被认为在类“中”。你的所有代码都出现在这里。默认情况下，脚本在类内包含两个方法：Start和Update：

```
// Use this for initialization
```

```
void Start (){
}
// Update is called once per frame
void Update (){
}
```

下一章中将更详细地介绍一些方法。目前，只需知道当场景第一次启动时，将运行放在Start方法内的任何代码。每次游戏更新时（大约每秒钟平均更新60次，这依赖于计算机），都将运行放在Update方法内的任何代码。

提示：

注释

程序设计语言可以让代码的作者为那些以后阅读代码的人留下一些信息，这些信息称为注释。两个正斜杠（//）后面的任何单词都将被“注释掉”，这意味着计算机将会忽略它们，而不会尝试把它们读作代码。在“Try It Yourself：创建脚本”中可以看到注释的示例。

注意：

Console（控制台）

到现在为止，Unity编辑器中还有一个窗口没有提到，即Console。实质上，Console 是一个包含游戏的文本输出的窗口。通常，当有一个来自脚本的错误或输出时，将把消息写到Console。图8.5显示了Console以及如何访问它。如果 Console 窗口不可见，也可以单击 Window >Console命令访问它。

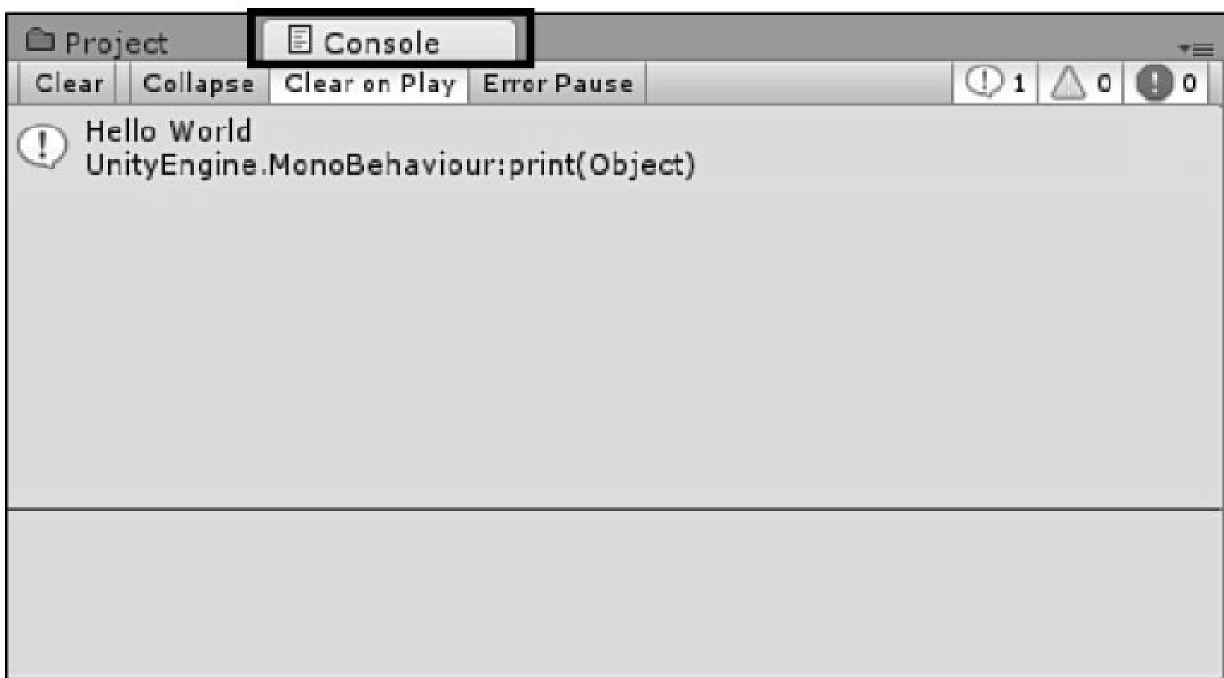


图8.5 Console窗口

使用内置的方法

让我们试验内置的方法Start 和Update，看看它们如何工作。在用于第8章（Hour 8）的本书配套资源中可以找到完成的ImportantFunctions脚本。尝试自己完成下面的练习，如果你感到困惑，可以参考本书配套资源。

（1）创建一个新的项目或场景，并向场景中添加一个名为ImportantFunctions的脚本。双击该脚本，打开MonoDevelop。

（2）在脚本内，向Start方法中添加下面一行代码：

```
print("Start runs before an object Updates");
```

（3）保存脚本，并在Unity中，把它附加到Main Camera 上。运行场景，并且注意出现在Console窗口中的消息。

（4）回到MonoDevelop中，并把下面一行代码添加到Update方法中：

```
print("This is called once a frame");
```

（5）保存脚本，并在 Unity 中快速开始和停止场景。注意：在

Console　　中，有一行文本来自Start方法，并有多行文本来自Update方法。

8.2 变量

有时，你希望在脚本中把相同的数据使用多次。你将需要该数据的占位符，并且可以重用它。这些占位符就称为变量（variable）。与传统的数学不同，程序设计中的变量可以包含的不仅只是数字。它们可以保存单词、复杂的对象或其他脚本。

8.2.1 创建变量

每个变量都有一个名称和一种类型，它们是在创建变量时提供的。可以利用下面的语法创建变量：

<variable type> <name>;

因此，要创建一个名为num1的整型变量，可以输入以下代码：

int num1;

表8.1包含了所有原始（或基本）变量类型以及它们可以保存的数据的类型。

注意：

语法

术语“语法”（syntax）指程序设计语言的规则。语法规定了如何构造和编写事物，使得计算机知道如何读取它们。你可能注意到我们的脚本中的每条语句或者每个命令都以分号结尾，这也是 C#语法的一部分。忘记分号将导致脚本不会工作。

表8.1 C#变量类型

类型	描述
int	integer（整型）的简写，用于存储正整数或负整数
float	float 存储浮点数据（比如 3.4），它是 Unity 中的默认数字类型
double	double 也存储浮点数，不过它不是 Unity 中默认数字类型。它一般保存比 float 更大的数字
bool	Boolean（布尔）的简写，用于存储“真”或“假”（在代码中实际上书写为 true 或 flase）
char	character（字符）的简写，用于存储单个字母、空格或特殊字符（比如 a、5 或!）。在书写字符值时，要带上单引号（'T'）
string	string 类型保存整个单词或句子。在书写字符串值时，要带上双引号（"Hello World"）

8.2.2 变量作用域

变量作用域指能够使用变量的地方。如你在脚本中看到的，类和方法使用开始和封闭大括号指示属于它们的内容。两个大括号之间的区域通常被称为块（block），这一点很重要，其原因是变量只能在创建它们的块中使用。因此，如果在脚本的 Start 方法内创建一个变量，将不能在 Update 方法中使用它。如果在不可使用变量的地方尝试使用它，则会导致一个错误。它们是两个不同的块。如果在类中但是在方法外面创建一个变量，那么两个方法都能够使用它，因为两个方法与变量位于相同的块（类块）中。程序清单8.2演示了这一点。

程序清单8.2 类和局部块级的演示

```
//This is in the "class block" and will
//be available everywhere in this class
private int num1;
void Start () {
    //this is in a "local block" and will
    //only be available in the Start method
    int num2;
}
```

8.2.3 公共和私有

如果查看程序清单8.2，将会在num1前看到关键字private。这称为访问修饰符（access modifier），只有在类级声明的变量才需要它。需要使用的访问修饰符有两个：private和public。关于这两个访问修饰符的内容有很多，你只需要知道它们在这个级别如何影响变量。实质上，私有变量（指带有关键字 private 的变量）只在创建它们的文件内有用，其他脚本和编辑器将不能看到它们或者以任何方式修改它们，它们只打算在内部使用。与之相反，公共变量对于其他脚本甚至Unity编辑器是可见的，这使得你很容易在Unity内自由地修改变量的值。

在Unity中修改公共变量

让我们看看在Unity编辑器中公共变量是如何可见的。

（1）创建一个新的C#脚本，然后在MonoDevelop中，在类中但是在Start方法上面添加下面一行代码：

```
public int runSpeed;
```

（2）保存脚本，然后在Unity中把它附加到Main Camera 上。

（3）选择Main Camera，并查看Inspector 视图。注意你刚才把脚本附加为一个组件。注意该组件具有一个新属性：Run Speed。可以在 Inspector 视图中修改该属性，并且所做的修改会在运行时反映在脚本中。如图 8.6 所示，查看具有新属性的组件。这幅图假定创建的脚本被命名为ImportantFunctions。

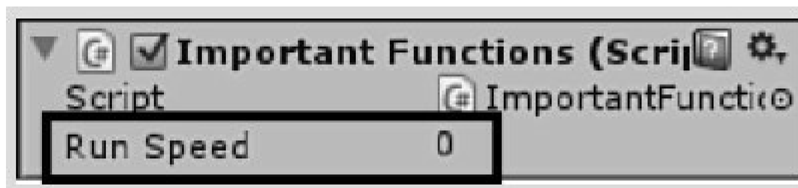


图8.6 脚本组件的新属性Run Speed

8.3 运算符

如果无法访问或修改变量，那么变量中的所有数据都将是无价值的。运算符是特殊的符号，使你能够对数据执行修改。它们一般属于以下4个类别之一：算术运算符、赋值运算符、相等性运算符和逻辑运算符。

8.3.1 算术运算符

算术运算符对变量执行某个标准的算术运算。它们一般只用于数字变量，尽管存在少数例外情况。表8.2描述了算术运算符。

表8.2 算术运算符

运算符	描述
+	加法。把两个数字相加起来。对于字符串，+符号用于把它们连接或结合在一起： "Hello"+"World";//produces"HelloWorld"
-	减法。用左边的数字减去右边的数字
*	乘法。把两个数字相乘起来
/	除法。用左边的数字除以右边的数字

%	求模。用左边的数字除以右边的数字，但是不返回结果。作为替代，求模运算将返回除法的余数：
	10% 2;//returns 0
	6% 5;//returns 1
	24% 7;//returns 3

算术运算符可以串联在一起，以产生更复杂的数学字符串：

x+(5*(6–y)/3);

算术运算符在工作时采用标准的算术运算顺序。从左到右执行运算，括号优先，乘法和除法次之，加法和减法再次之。

8.3.2 赋值运算符

赋值运算符就像听上去的那样，它们用于给变量赋值。最著名的赋值运算符是等于号，但是更常见的是把多个运算结合在一起。C#中的所有赋值都是把右边的值赋予左边，这意味着将把右边的任何内容都移到左边：

x = 5; //This works. It sets the variable x to 5.

5 = x; //This does not work. You cannot assign a variable to a value (5).

表8.3描述了赋值运算符。

表8.3 赋值运算符

运算符	描述
=	把右边的值赋予左边的变量

+=、-=、*=、/=	简写的赋值运算符，基于使用的符号执行某种算术运算，然后把结果赋予左边的任何变量：
	x = x + 5; //Adds 5 to x and then assigns it to x
	x += 5; //Does the same as above, only shorthand

++、--	另一类简写的运算符，它们称为递增和递减运算符，分别用于把一个数字增加或减小 1：
	x = x + 1; //Adds 1 to x and then assigns it to x
	x++; //Does the same as above, only shorthand

8.3.3 相等性运算符

相等性运算符用于比较两个值。相等性运算符的结果总是“真”或“假”。因此，可以保存相等性运算符的结果的唯一变量类型是布尔型（记住：布尔型只能包含“真”或“假”）。表8.4描述了相等性运算符。

表8.4 相等性运算符

运算符	描述

==	不要把它与赋值运算符（=）弄混淆，仅当两个值相等时，它才返回“真”。否则，它将返回“假”：
	`5 == 6; //Returns false`
	`9 == 9; //Returns true`

>、<	它们是“大于”和“小于”运算符：
	`5 > 3; //Returns true`
	`5 < 3; //Returns false`

>=、<=	它们类似于“大于”和“小于”，只不过它们是“大于或等于”和“小于或等于”运算符：
	`3 >= 3; //Returns true`
	`5 <= 9; //Returns true`

!=	这是“不等于”运算符，仅当两个值不同时，它才返回“真”。否则，它将返回“假”：
	`5 != 6; //Returns true`
	`9 != 9; //Returns false`

提示：

额外的练习

在用于第8章（Hour 8）的本书配套资源中，有一个名为EqualityAnd Operations.cs的脚本。一定要仔细查看它，对不同的运算符进行一些额外的练习。

8.3.4 逻辑运算符

逻辑运算符使你能够把两个或更多的布尔值（“真”或“假”）结合成单个布尔值，它们可用于确定复杂的条件。表8.5描述了逻辑运算符。

表8.5 逻辑运算符

<table>
<tr><th>运算符</th><th>描述</th></tr>
<tr><td>&&</td><td>它称为逻辑“与”运算符，用于比较两个布尔值，并确定它们是否都为“真”。如果两个值中的任何一个值为“假”或者都为“假”，它将返回“假”：
<code>true && false; //Returns false</code>
<code>false && true; //Returns false</code>
<code>false && false; //Returns false</code>
<code>true && true; //Returns true</code></td></tr>
<tr><td>||</td><td>它称为逻辑“或”运算符，用于比较两个布尔值，并确定其中是否有任何一个值为“真”。如果其中任何一个值为“真”或者两个值都为“真”，它将返回“真”：
<code>true && false; //Returns true</code>
<code>false && true; //Returns true</code>
<code>false && false; //Returns false</code>
<code>true && true; //Returns true</code></td></tr>
<tr><td>!</td><td>它称为逻辑“非”运算符，用于返回一个布尔值的相反值：
<code>!true; //Returns false</code>
<code>!false; //Returns true</code></td></tr>
</table>

8.4 条件语句

计算机的大部分能力都在于它能够做出初步的决定，这种能力的根源在于布尔“真”和“假”。可以使用这些布尔值来构建条件语句，并且指导程序沿着独特的方向前进。在通过代码构建流程或逻辑时，只需记住机器一次只能做出一个简单的决定。不过，把足够多的决定放在一起，就能构建出复杂的交互。

8.4.1 if 语句

条件语句的基础是if语句，其构造如下：

```
if( <some Boolean condition>)
{
   //do something
}
```

if结构可以读用“如果它为‘真’，就执行它”。因此，如果你希望在x大于5时把“Hello World”输出到Console，就可以编写如下代码：

```
if(x > 5)
{
   print("Hello World");
}
```

记住：if语句条件的内容必须求值为“真”或“假”。把数字、单词或其他任何内容放在那里都将不会工作：

```
if( "Hello" == "Hello"）//Correct
{}
```

```
if( x + y）//Incorrect
{}
```

最后，当条件求值为“真”时，你希望运行的任何代码都必须放在if语句后面的开始和封闭大括号内。

提示：

古怪的行为

条件语句使用一种特定的语法，如果你没有准确遵循它，那么它可能会展示奇怪的行为。你可能发现在代码中具有一个if语句，并且发现某个部分不是非常正确。可能条件代码一直都会运行，甚至当它不应该运行时亦会如此。你也可能发现它永远都不会运行，甚至当它应该运行时也是如此。你需要知道两种常见的原因：第一，i　f条件后面没有分号，如果编写带有分号的 if语句，则总会运行其后的代码；第二，确保在　if语句内使用相等性运算符（==），而不是赋值运算符（=），否则可能会导致怪异的行为：

```
if(x > 5); //Incorrect
if(x = 5); //Incorrect
```

8.4.2 if/else语句

if 语句非常适合于条件代码，但是如果想把程序分支到两条不同的路径，则该怎么办？if/else语句将使你能够这么做。if/else语句具有与if语句相同的基本前提，只不过它可以读作“如果这个条件为‘真’，就做这件事情，否则就做另外一件事情”。if/else语句可以写成如下形式：

```
if( <some Boolean condition>)
{
  //Do something
}
```

```
else
{
   //Do something else
}
```

例如，如果希望在变量x 大于变量y时把“X is greater than Y”打印到Console，或者希望在x 不大于y 时打印“Y is greater than X”，可以编写以下语句：

```
if(x > y)
{
   print("X is greater than Y");
}
else
{
   print("X is greater than Y");
}
```

8.4.3 if / else if 语句

有时，你想要代码分支到多条路径之一。你可能希望用户能够从一组选项（比如一个菜单）中选择一个选项。if /else if 的构造方式与前两种结构非常相似，只不过它具有多个条件：

```
if( <some Boolean condition>)
{
   //Do something
}
else if( <some other Boolean condition>)
{
```

```
    //Do something else
}
else //The else is optional in the IF / ELSE IF statement
{
    //Do something else
}
```

例如，如果希望基于一个人的百分制分数把他的字母等级输出到控制台，可以编写如下代码：

```
if(grade >= 90)
{
    print("You got an A");
}
else if(grade >= 80)
{
    print("You got a B");
}
else if(grade >= 70)
{
    print("You got a c");
}
else if(grade >= 60)
{
    print("You got a D");
}
else
{
    print("You got an F");
```

```
}
```

提示：

单行if语句

如果if语句只包含单独一行代码，将不需要具有开始和封闭大括号。因此，代码可以写成如下形式：

```
if(x > y)
{
    print("X is greater than Y");
}
```

也可以写成如下形式：

```
if(x > y)
    print("X is greater than Y");
```

8.5 迭代

迄今为止，你已经学习了如何处理变量，当你想要做像把两个数字相加起来这样的事情时，这当然是有用的。但是，如果你想把1～100之间的所有数字相加起来，则该如何？1～1000 之间呢？你肯定不想输入所有的代码。作为替代，可以使用称为迭代（iteration）的结构（通常也称之为循环[looping]）。可以使用两种主要的循环类型：while 循环和for循环。

8.5.1 while循环

while循环是最基本的迭代形式，它遵循与if语句类似的结构：

```
while(<some Boolean condition>)
{
   //do something
}
```

它们之间的唯一区别是：if 语句只会把它的包含代码运行一次，而循环将反复运行它的包含代码，直到条件变“假”为止。因此，如果想把1～100之间的所有数字相加起来，然后把它们输出到控制台，可以编写如下代码：

```
int sum = 0;
int count = 1;
while(count <= 100)
{
   sum += count;
```

```
    count++;
}
print(sum);
```

可以看到，count的值将从1开始，并且每次迭代或者每执行循环一次都会将其增加1，直至它等于101为止。当count等于101时，它将不再小于或等于100，并将退出循环。省略count++这一行将导致循环无限地运行（因此一定要包括这一行代码）。在循环的每次迭代期间，都会把count的值加到变量sum上。一旦循环退出，就把和写到控制台上。

总之，只要while循环的条件为“真”，它就会反复运行所包含的代码。一旦它的条件变为“假”，它就会停止循环。

8.5.2 for循环

for循环遵循与while循环具有相同的思想，只不过它的构造方式稍有不同。如你在前面用于while循环的代码中所看到的，必须创建一个count变量，必须测试变量（作为条件），还必须增加变量，这些都是在3个单独的行上完成的。for循环把该语法精简成单独一行，它看起来如下：

```
for(<create a counter> ; <Boolean conditional> ; <increment the counter >)
{
    //Do something
}
```

for循环具有3个特殊的间隔部分（compartment）用于控制循环。注意for循环头部中的每个区域之间的分号，而不是逗号。第一个间隔部分创建一个用作计数器的变量（用于计数器的常见名称是i，即iterator（迭代器）的简写）；第二个间隔部分是循环的条件语句；第三个间隔部分

用于增加或减少计数器。可以使用for循环重写前面的while循环示例，如下所示：

```
int sum = 0;
for(int count = 1; count <= 100; count++)
{
    sum += count;
}
print(sum);
```

如你所看到的，可以对循环的不同部分进行精简，使之占据较少的空间。可以看到 for循环确实擅长处理像计数这样的事情。

8.6 小结

在本章中，你初次涉足了视频游戏编程。你首先了解了在Unity中编写脚本的基础知识，接着学习了如何创建和附加脚本，还看到了关于脚本的基本分析。然后，你学习了程序的基本逻辑成分，接着认识了变量、运算符、条件语句和循环语句。

8.7 问与答

问：制作游戏需要多少程序设计工作量？

答：大多数游戏都会使用某种形式的程序设计来定义复杂的行为。行为需要越复杂，程序设计就需要越复杂。如果想要制作游戏，肯定应该熟悉程序设计的概念。即使你不打算成为游戏的主要开发人员也是如此。考虑到这一点，本书将介绍你制作自己的前几款简单游戏所需知道的一切知识。

问：本书中介绍了关于编写脚本的所有知识吗？

答：是又不是。本书中展示了基本的程序设计，它们永远也不会真正改变，而只会以新的、独特的方式应用它们。也就是说，由于一般意义上的程序设计的复杂性，这里介绍的许多内容都进行了简化。如果你想学习关于程序设计的更多知识，应该阅读专门介绍这个主题的图书或文章。

8.8 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

8.8.1 问题

1．Unity允许利用哪3种语言编写程序？

2．判断题：Start方法中的代码在每一帧处运行。

3．在Unity中，哪种变量类型是默认的浮点数类型？

4．哪个运算符返回除法的余数？

5．什么是条件语句？

6．哪种循环类型最适合于计数？

8.8.2 答案

1．C#、JavaScript和Boo。

2．错误。Start方法在场景的开始处运行；Update方法则在每一帧处都会运行。

3．float。

4．求模运算符。

5．条件语句是一种允许计算机基于简单的决定选择代码路径的代码结构。

6．for循环。

8.8.3 练习

把编码结构视作构件是有益的。其中每个部分单独来看都比较简

单。不过，把它们放在一起，就可以构建复杂的实体。后文中，还将会有多个编程挑战。使用你在本章中获得的知识，构建问题的解决方案。把每个解决方案都放在它自己的脚本中，并把脚本附加到场景的Main Camera上，确保它们都会工作。在用于第8章（Hour 8）的本书配套资源中可以找到这个练习的解决方案。

1．编写一个脚本，把2～499之间的所有偶数相加起来，并把结果输出到控制台上。

2．编写一个脚本，把1～100之间的所有数字输出到控制台上，但是不要输出3或5的倍数，并代之以输出“Programming is Awesome!”（提示：如果求模运算的结果是0，就可以判断一个数字是否是另一个数字的偶数）。

3．在斐波纳契数列中，通过把前两个数字相加来确定下一个数字。该数列开始于0、1、1、2、3、5……编写一个脚本，确定斐波纳契数列的前20个数字，并把它们输出到控制台上。

第9章 编写第2部分的脚本

在本章中你将学到：

怎样编写方法；

怎样捕获用户输入；

怎样处理局部组件；

怎样处理游戏对象。

在上一章中，你学习了在 Unity 中编写脚本的基础知识。在本章中，你将使用所学的知识完成更有意义的任务。你首先将研究方法，学习它们是什么、它们如何工作以及如何编写它们。然后，你将处理用户输入。之后，将研究如何从脚本中访问组件。在本章最后，将学习如何利用代码访问其他游戏对象以及它们的组件。

提示：

示例脚本

用于第9章（Hour 9）的本书配套资源提供了本章中提到的多个脚本和编码结构，一定要检查它们，以进行额外的学习。

9.1 方法

方法（通常称为函数，function）是可以调用并且彼此独立使用的代码模块。每个方法一般都代表单独的任务或目的，通常许多方法可以协同工作以实现复杂的目标。考虑你迄今为止见过的两个方法：Start和Update，其中每个方法都代表一个单独、简明的目的。Start方法包含在第一次开启场景时为对象运行的所有代码，Update方法则包含在场景的每一帧处都会运行的代码。

注意：

方法的简写形式

到目前为止，你已经看到：无论何时提及Start方法，在它后面都会接着“方法”一词。总是不得不使用一个词语指定它是一个方法可能会令人觉得麻烦，不过，不能只书写Start，因为这样做的话人们将不知道你所指的是一个单词还是一个方法。处理这种情况的一个更简洁的方式是在单词后面添加圆括号。因此，Start方法也可以写为Start( )。如果你看到书写为像SomeWords( )这样的内容，那么立即就可以知道书写者谈论的是一个名为SomeWords的方法。

9.1.1 方法的具体分析

在使用方法之前，应该探讨一下组成它们的不同部分。下面给出了方法的一般形式：

```
<return type> <name>(<parameters>)
{
   <Inside the method's block>
```

}

1．方法名

每个方法都必须具有一个独特的名称。尽管影响正确名称的规则是由使用的语言确定的，针对方法名的良好的一般指导原则如下。

使方法名具有描述性。它应该是一个动作，最好是一个动词。

避免在方法名中出现空格。不允许使用空格。

避免在方法名中出现特殊字符（!、@、*、%、$等）。不同的语言允许不同的字符。不使用任何特殊字符，就不会冒着出现问题的风险。

方法名很重要，因为方法名既用于标识方法，又说明了如何使用它们。

2．返回类型

每个方法都有能力返回一个变量给任何调用它的代码。这个变量的类型就称为返回类型（return type）。如果方法返回一个整数，返回类型就是 int。同样，如果方法返回一个“真”值或“假”值，返回类型就是bool。如果方法不返回任何值，那它仍然具有一种返回类型，在这种情况下，返回类型是 void（意指不返回任何内容）。任何返回一个值的方法都将利用关键字return来执行该任务。

3．参数列表

就像方法可以把一个变量传回任何调用它的代码一样，调用代码也可以传入变量，这些变量就称为参数（parameter）。传入方法的变量是在方法的参数列表部分标识的。接受一个整数enemyID的名为Attack的方法的示例如下：

```
void Attack(int enemyID)
{}
```

可以看到，在指定参数时，必须同时提供变量类型以及名称。用逗号把多个参数分隔开。

4．方法块

这是方法的代码实际出现的位置。每次使用一个方法时，将作为一个整体运行方法块内的代码。

确定方法的各个部分

花一点时间检查方法的不同部分。给定下面的方法：

```
int TakeDamage(int damageAmount)
{
   int health = 100;
   return health–damageAmount;
}
```

你能够确定下面各个部分吗？

1．方法的名称是什么？

2．方法返回的变量类型是什么？

3．方法参数是什么？有多少个参数？

4．方法的块中的代码是什么？

提示：

将方法视作工厂

对于程序设计初学者来说，方法的概念可能令人混淆。通常，可能会在方法的参数和返回值上犯错。使之保持直观的一种良好的方式是把方法视作工厂。工厂接收原材料，并使用它们制作产品。方法也一样。参数是你传入“工厂”的材料，返回值是那个工厂的最终产品。可以把不带参数的方法视作不需要原材料的工厂。同样，可以把不返回任何内容的方法视作不制造最终产品的工厂。通过把方法想象成小型工厂，可以在你的头脑中努力使逻辑的流程保持直观。

9.1.2 编写方法

既然你已经理解了方法的成分，编写它们就很容易。在开始编写方

法之前，花一点时间问自己3个主要的问题。

1．方法将完成的特定任务是什么？

2．方法为完成任务而需要任何外部数据吗？

3．方法需要返回任何数据吗？

回答这些问题将有助于确定方法的名称、参数和返回数据。

考虑下面这个示例：玩家被一只火球击中。你需要编写一个方法，通过去除5个生命值来模拟这种情况。你知道这个方法的特定任务是什么，还知道该任务不需要任何数据（因为你知道它要花费5个生命值），并且可能应该返回新的生命值。可以像下面这样编写该方法：

```
int TakeDamageFromFireball()
{
   int playerHealth = 100;
   return playerHealth–5;
}
```

在这个方法中可以看到，玩家的生命值是100，并从中减去了5个生命值。然后传回结果（它是95）。显然，可以改进它。对于初学者，上面所说的是火球造成了5个生命值的伤害，但是如果你希望它造成更大的伤害，则该如何？你将需要准确知道在任何给定的时间，指望一只火球造成多大的伤害。你将需要一个变量，或者在这里是一个参数。新方法可以编写成如下状态：

```
int TakeDamageFromFireball(int damage)
{
   int playerHealth = 100;
   return playerHealth–damage;
}
```

现在可以看到伤害是从方法读入的，并且应用于玩家的生命值。可以改进的另一个位置是生命值本身。目前，玩家可能永远也不会失去生

命，因为在去除伤害之前总会把他们的生命值恢复到 100。把玩家的生命值存储在别的位置将会更好，使得它的值将会是持久的。然后可以读入它，并且相应地消除伤害。这样，方法将如下所示：

```
int TakeDamageFromFireball(int damage, int playerHealth)
{
   return playerHealth–damage;
}
```

通过检查你的需求，可以为游戏构建更好、更健壮的方法。

注意：

简化

在上面的示例中，得到的方法将简单地执行基本的减法运算。出于指导的缘故，这过于简单化。在更现实的环境中，有许多方式可以处理这个任务。可以把玩家的生命值存储在一个属于脚本的变量中。这样做将意味着不需要读入它。另一种可能性是 TakeDamageFromFireball 方法中的一个复杂的算法，其中将通过某个盔甲值、玩家的躲避能力或者某个魔法盾减小传入的伤害。如果你觉得这里的示例看上去有些愚蠢，只需记住它们只用于演示所介绍的主题的不同元素。

1．使用方法

一旦编写了方法，剩下的事就是使用它。使用方法通常称为调用（call或invoke）方法。要调用一个方法，只需写出方法的名称，其后接着圆括号和任何参数。因此，如果尝试使用一个名为SomeMethod的方法，只需编写以下代码：

```
SomeMethod();
```

如果SomeMethod()需要一个整型参数，可以像下面这样调用它：

```
//Method call with a value of 5
SomeMethod（5）;
//Method call passing in a variable
```

```
int x = 5;
SomeMethod(x); //do not write "int x" here.
```

注意：

在调用一个方法时，不需要与传入的变量一起提供变量类型。如果SomeMethod( )返回一个值，将希望在一个变量中捕获（catch）它。代码可能看起来如下（假定采用布尔返回类型；实际上，它可以是任何类型）：

```
bool result = SomeMethod();
```

编写方法只需使用这种基本的语法。

调用方法

让我们进一步处理上一节中描述的TakeDamageFromFireball方法。在这个练习中，将调用该方法的多种形式。在用于第9章（Hour 9）的本书配套资源中可以找到这个练习的解决方案，即FireBallScript。

（1）创建一个新的项目或场景。在用于第9章（Hour 9）的本书配套资源中定位FireBallScript，并把它导入到项目中。此外，也可以创建一个名为FireBallScript的C#脚本，并且输入前面描述过的3个TakeDamageFromFireball方法。

（2）在Start方法中，输入以下代码，调用第一个TakeDamageFromFireball()：

```
int x = TakeDamageFromFireball();
print ("Player health: " + x);
```

（3）把脚本附加到Main Camera 上，并运行场景。注意控制台中的输出。现在，在Start()中输入以下代码（把它放在你输入的第一段代码之下，而无需删除这段代码），调用第二个TakeDamageFromFireball()：

```
int y = TakeDamageFromFireball(25);
print ("Player health: " + y);
```

（4）再次运行场景，并且注意控制台中的输出。最后，在 Start()中输入以下代码，调用最后一个TakeDamageFromFireball()：

```
int z = TakeDamageFromFireball(30, 50);
print ("Player health: " + z);
```

（5）运行场景，并且注意最终的输出。看看全部3个方法的行为有何不同。要特别注意你是怎样调用每个方法的。

9.2 输入

如果没有玩家的输入，视频游戏（video game）将只是视频（video）。玩家输入可以具有许多不同的类型。输入可以是物理的，比如游戏手柄、游戏操纵杆、键盘和鼠标；有电容式控制器，比如在现代移动设备中发现的相对较新的触摸屏；还有移动设备，比如Wii Remote、PlayStation Move 和Microsoft Kinect。较少见的是音频输入，它使用麦克风和玩家的语音控制游戏。在本节中，你将学习编写代码，允许玩家利用物理设备与游戏交互。

9.2.1 输入的基础知识

利用Unity（像大多数游戏引擎一样），可以在代码中检测特定的按键，使之成为交互式的。不过，避免这样做将是一个好主意。因为这样将使玩家难以将控制重新映射到他们的偏好上。幸好Unity具有一个简单的系统，可以优雅地映射控制。利用Unity，你将寻找一根特定的轴（axis），以便知道玩家是否想要执行某个动作。然后，当玩家运行游戏时，他就可以选择使不同的控制意指不同的轴。

可以使用Input Manager查看、编辑和添加不同的轴。要访问Input Manager，可以单击Edit > Project Settings > Input 命令。在Input Manager中，可以查看与不同的输入动作关联的多根轴。默认情况下，有15根输入轴，但是如果需要，也可以添加你自己的输入轴。图9.1显示了具有展开的水平轴的默认Input Manager。

▼ Horizontal	
Name	Horizontal
Descriptive Name	
Descriptive Negative Name	
Negative Button	left
Positive Button	right
Alt Negative Button	a
Alt Positive Button	d
Gravity	3
Dead	0.001
Sensitivity	3
Snap	☑
Invert	☐
Type	Key or Mouse Button
Axis	X axis
Joy Num	Get Motion from all Joysticks

图9.1 Input Manager

虽然水平轴不会直接控制一切（我们以后将编写脚本来执行该任务），但它代表横向行走的玩家。表9.1描述了轴的属性。

表9.1 轴的属性

属性	描述
Name	轴的名称，在代码中利用该属性引用轴
Descriptive Name / Descriptive Negative Name	游戏配置中在玩家看来一个有些啰唆的轴名称。负名称指相反的名称。例如：“Go left”和“Go right”就是名称和负名称对
Negative Button / Positive Button	给轴传递负值和正值的按钮。对于水平轴，它们是左箭头和右箭头
Alt Negative Button / Alt Positive Button	给轴传递值的替换按钮。对于水平轴，它们是 A 键和 D 键
Gravity	一旦不再按键，轴将多快地回归到 0
Dead	比这个值小的任何输入都将被忽略。这有助于防止游戏操纵杆设备抖动
Sensitivity	轴将多快地响应输入
Snap	当选中这个属性时，如果按下相反的方向，它将导致轴立即回归到 0
Invert	当选中这个属性时，将使控制发生颠倒
Type	输入的类型，包括键盘/鼠标键、鼠标移动和游戏操纵杆移动
Axis	来自输入设备的对应轴，这不适用于按钮
Joy Num	指示从哪根游戏操纵杆获取输入，默认将从所有的游戏操纵杆获取输入

9.2.2 编写输入脚本

一旦在Input Manager中设置了轴，在代码中处理它们就很简单。要访问玩家的任何输入，都将使用Input对象。更确切地讲，将使用输入对象的GetAxis方法。GetAxis( )将读入轴的名称作为字符串，并且返回那根轴的值。因此，如果希望获得水平轴的值，可以输入以下代码：

```
float hVal = Input.GetAxis("Horizontal");
```

对于水平轴，如果玩家按下左箭头键（或A键），GetAxis( )将返回一个负数。如果玩家按下右箭头键（或D键），该方法将返回一个正数。

读入用户输入

让我们使用垂直轴和水平轴，以更好地理解如何使用玩家输入。

（1）创建一个新的项目或场景，向项目中添加一个名为PlayerInput的脚本，并把它附加到Main Camera 上。

（2）把以下代码添加到PlayerInput脚本中的Update方法中：

```
float hVal = Input.GetAxis("Horizontal");
float vVal = Input.GetAxis("Vertical");
if(hVal != 0)
    print("Horizontal movement selected: " + hVal);
if(vVal != 0)
    print("Vertical movement selected: " + vVal);
```

（3）保存脚本并运行场景，注意当按下箭头键时所发生的事情。现在试试按下W键、A键、S键和D键。

9.2.3 特定的键输入

尽管你一般希望处理输入的通用轴，有时也会希望确定是否按下了

特定的键。为此，将再次使用这个输入对象。不过，这一次将使用GetKey　方法。该方法读入与特定键对应的特殊代码，然后如果当前按下了这个键，它就返回一个“真”值；如果当前没有按下这个键，它就返回一个“假”值。为了确定当前是否按下了K键，可以输入以下代码：

```
bool isKeyDown = Input.GetKey(KeyCode.K);
```

提示：

查找键码

每个键都具有特定的键码，通过阅读Unity文档，可以确定你想要的特定键的键码。此外，也可以使用MonoDevelop的内置工具来查找它。无论何时在MonoDevelop中处理脚本，总是可以输入一个对象的名称，其后接着一个点号。这样做将导致一个菜单弹出，其中带有所有可能的选项。同样，如果在输入方法名后接着输入一个开始圆括号，将弹出相同的菜单，并且显示多个选项。图9.2说明了如何使用自动菜单查找Esc键的键码。

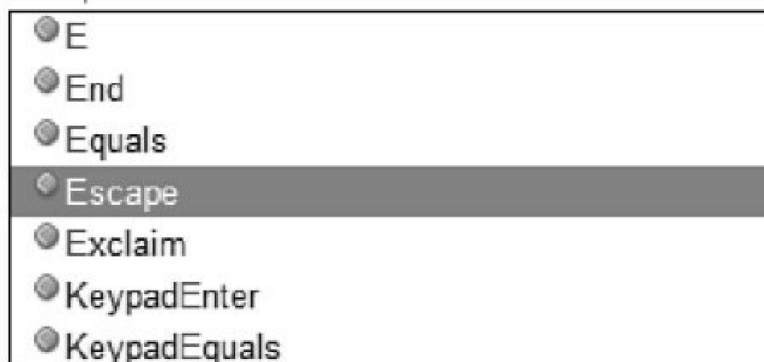


图9.2 MonoDevelop 中自动弹出的菜单

读入特定的按键

让我们编写一个脚本，确定是否按下了特定的键。

（1）创建一个新的项目或场景，向项目中添加一个名为PlayerInput的脚本，并把它附加到Main Camera 上。

（2）把以下代码添加到PlayerInput脚本中的Update方法中：

```
if(Input.GetKey(KeyCode.M))
   print("The 'M' key is pressed down");
```

（3）保存脚本并运行场景。注意当按下M键时所发生的事情。

9.2.4 鼠标输入

除了按键之外，你还希望捕获用户的鼠标输入。有两个组件用于鼠标输入：鼠标键和鼠标移动。确定是否按下了鼠标键与前面介绍的按键非常相似，你将再次使用 Input 对象。这一次，将使用GetMouseButtonDown方法，该方法接受一个1～3之间的整数，指定你正在探讨哪个鼠标键。该方法返回一个布尔值，指示是否按下了鼠标键。用于获取鼠标按键的代码如下所示：

```
bool isButtonDown;
isButtonDown = Input.GetMouseButtonDown(0); //left mouse button
isButtonDown = Input.GetMouseButtonDown(1); //right mouse button
isButtonDown = Input.GetMouseButtonDown(3); //center mouse button
```

鼠标移动只沿着两根轴（x轴和y轴）进行。要获取鼠标移动，可以使用输入对象的GetAxis方法。可以使用名称Mouse X 和Mouse Y 分别获取沿着x 轴和y 轴的移动。用于读入鼠标移动的代码将如下所示：

```
float value;
value = Input.GetAxis("Mouse X"); //x axis movement
value = Input.GetAxis("Mouse Y"); //y axis movement
```

与鼠标按键不同，鼠标移动是通过仅仅自上一帧起鼠标移动的距离测量的。实质上，按住一个键将导致某个值增加，直至它到达−1或1的上限为止（依赖于它是正数还是负数）。不过，鼠标移动一般具有较小的数字，因为每个帧都会测量和重置它。

读取鼠标移动

在这个练习中，将读入鼠标移动，并把结果输出到Console。

（1）创建一个新的项目或场景，向项目中添加一个名为PlayerInput的脚本，并把它附加到Main Camera 上。

（2）把以下代码添加到PlayerInput脚本中的Update方法中：

```
float mxVal = Input.GetAxis("Mouse X");
float myVal = Input.GetAxis("Mouse Y");
if(mxVal != 0)
   print("Mouse X movement selected: " + mxVal);
if(myVal != 0)
   print("Mouse Y movement selected: " + myVal);
```

（3）保存脚本并运行场景。仔细检查控制台，看看当你四处移动鼠标时的输出结果。

9.3 访问局部组件

正如你在Inspector视图中见过许多次的，对象由多个组件组成。你可以在运行时通过脚本与这些组件交互。每个组件都有所不同，但是用于编辑组件的一般语法是输入组件的名称，其后接着一个点号，并用你想修改的属性名称结尾。例如，如果想更改点光源组件的类型，可以编写如下代码：

```
light.type = LightType.Directional;
```

这个语法把灯光组件的类型属性更改为定向灯光。注意灯光组件和类型属性在Inspector中是大写的，但是在代码中是小写的。只需记住：当你尝试访问特定的事物（例如，“这个灯光”）时，可以使用小写字母。

你将处理的最常见的组件是变换组件。通过编辑它，可以使对象在屏幕上四处移动。记住：对象的变换由它的平移（或位置）、旋转和缩放组成。尽管可以直接修改它们，但是使用一些内置的选项（称为Translate 方法、Rotate 方法和 localScale 变量）将更容易：

```
//Moves the object along the positive x axis.
//The '0f' means 0 as a float. It is the way Unity reads floats
transform.Translate(.05f, 0f, 0f);
//Rotates the object along the z axis
transform.Rotate(0f, 0f, 1f);
//Scales the object to double its size in all directions
transform.localScale = new Vector3(1.5f, 1.5f, 1.5f);
```

由于Translate(　)和Rotate()是方法，如果把上面的代码放在Update()中，对象将持续沿着正x轴移动，同时沿着y轴旋转。

变换对象

让我们把前面的代码应用于场景中的对象，看看它的实际作用。

（1）创建一个新的项目或场景，向场景中添加一个立方体，并把它定位于(0, −1, 0)处。

（2）创建一个新的脚本，并把它命名为 CubeScript。把该脚本置于立方体上。在MonoDevelop中，把以下代码输入到Update方法中：

```
transform.Translate(.05f, 0f, 0f);
transform.Rotate(0f, 0f, 1f);
transform.localScale = new Vector3(1.5f, 1.5f, 1.5f);
```

（3）保存脚本并运行场景。注意Translate和Rotate方法的作用是累积的，而localScale变量则不然，它不会保持增长。

9.4 访问其他对象

许多次，你都希望脚本能够查找和操纵其他对象以及它们的组件。这是一件很简单的事情，只需查找你想要的对象并调用合适的组件即可。可以用几种基本的方式查找非脚本局部所有的对象，或者查找附加脚本的对象。

9.4.1 查找其他对象

查找其他对象，以便处理它们的第一种并且是最容易的方式是使用编辑器。通过在类级创建一个GameObject类型的公共变量，可以在Inspector视图中把想要的对象简单地拖到脚本组件上。用于设置它的代码如下所示：

```
//This is here for reference
    public class SomeClassScript : MonoBehaviour {
        //This is the game object you want to access
        public GameObject objectYouWant;
        //This is here for reference
        Void Start(){
        }
    }
```

在把脚本附加到游戏对象上之后，在 Inspector 中可以看到一个名为 Object You Want的属性，如图 9.3所示。只需把你想要的任何游戏对象拖到这个属性上，即可在脚本中访问它。

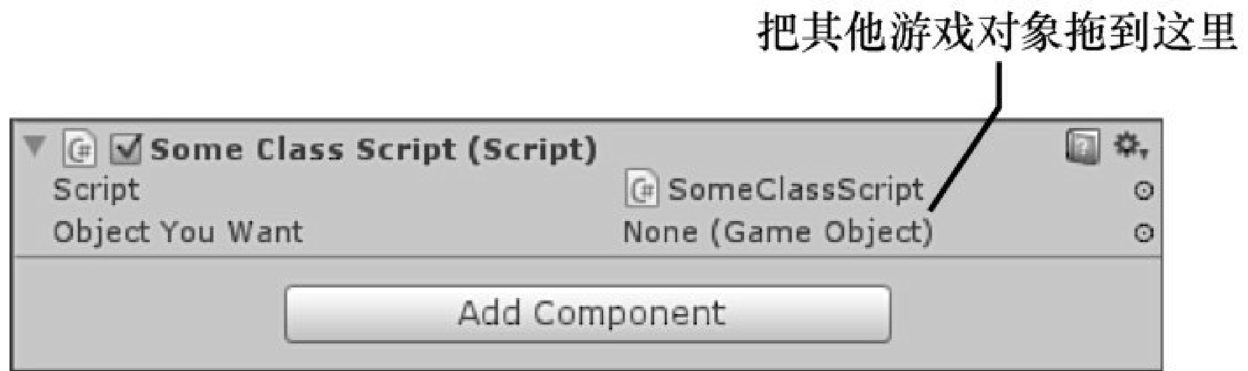


图9.3 Inspector 中新出现的 Object You Want属性

查找游戏对象的另一种方式是使用Find方法。要利用这种方式查找它，将需要知道对象的名称。对象的名称是出现在Hierarchy视图内的名称。假定你正在寻找一个名为Cube的对象，代码将如下所示：

```
//This is here for reference
public class SomeClassScript : MonoBehaviour {
   //This is the game object you want to access
   public GameObject target;
   //This is here for reference
   void Start(){
      target = GameObject.Find("Cube");
   }
}
```

这种方法的缺点是：它只会返回所查找的第一个具有给定名称的项目。如果具有多个Cube对象，将不知道获得的是哪个对象。

查找对象的最后一种方式是利用它的标签进行查找。对象的标签非常像它的图层（在前面介绍过图层），唯一的区别是语义。图层用于广泛的交互类型，而标签则用于基本的标识。可以使用Tag Manager（单击Edit > Project Settings > Tags 命令）创建标签。如图9.4所示，显示了如何向Tag Manager 中添加新标签。

一旦创建了标签，只需在Inspector视图中使用Tab下拉列表把它应用于对象即可，如图9.5所示。



图9.4 添加新标签

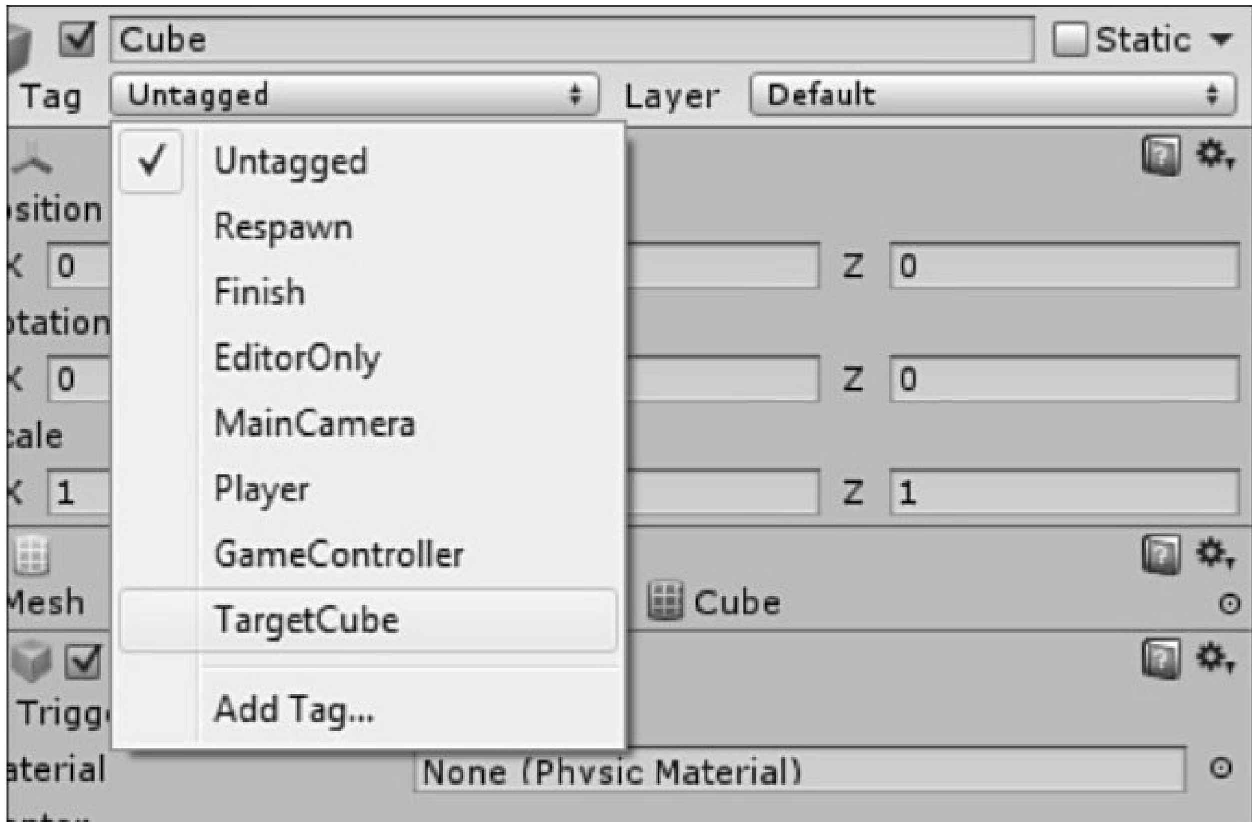


图9.5 选择标签

既然已经给对象添加了标签，就可以使用FindWithTag方法查找它：

```
//This is here for reference
public class SomeClassScript : MonoBehaviour {
   //This is the game object you want to access
   public GameObject target;
   //This is here for reference
   void Start(){
      target = GameObject.FindWithTag("TargetCube");
   }
}
```

提示：

查找效率

在前面的示例中，目标游戏对象存储在一个类变量（通常称为成员[member]）中。用于查找目标对象的代码放在Start方法中。总是可以简单地创建变量，并在Update方法（或者需要它的其他位置）中查找目标对象，但是应该避免这样做。反复查找对象是效率低下的，并且可能减慢游戏的性能。记住：性能最重要。反复做同样的事情很糟糕。

9.4.2 修改对象组件

一旦具有指向另一个对象的引用，处理该对象的组件就几乎100%相同。唯一的区别是：现在，无需简单地书写组件名称，而只需书写对象变量，并在它前面输入一个点号即可：

```
//This accesses the local component, not what you want
transform.Translate(0, 0, 0);
//This accesses the target object, what you want
targetObject.transform.Translate(0, 0, 0);
```

变换目标对象

让我们花点时间使用脚本修改一个目标对象。

（1）创建一个新的项目或场景，向场景中添加一个立方体，并把它定位于(0, –1, 0)处。

（2）创建一个新脚本，并把它命名为TargetCubeScript。把该脚本放在Main Camera 上。在MonoDevelop中，把以下代码输入到TargetCubeScript中：

```
//This is the game object you want to access
public GameObject target;
//This is here for reference
void Start(){
  target = GameObject.Find("Cube");
```

```
}
void Update(){
    target.transform.Translate(.05f, 0f, 0f);
    target.transform.Rotate(0f, 0f, 1f);
    target.transform.localScale = new Vector3(1.5f, 1.5f, 1.5f);
}
```

（3）保存脚本并运行场景。注意立方体在四处移动，即使脚本是应用于Main Camera的。

9.5 小结

在本章中，你探索了在Unity中编写脚本的更多知识。你学习了方法，并且探讨了编写你自己的方法的一些方式。然后，你处理了来自键盘和鼠标的玩家输入。之后，你学习了利用代码修改对象组件。在本章最后，你学习了如何查找其他游戏对象，并通过脚本与它们交互。

9.6 问与答

问：我应该编写多少个方法?

答：方法应该是单个简洁的函数。方法不应太少，因为这将导致每个方法做不止一件事情。你也不希望方法太多，因为这将违背自己的初衷。只要每个过程都具有它自己特定的方法，就足够了。

问：我们为什么没有学习关于游戏手柄的更多知识?

答：游戏手柄的问题在于它们全都有所不同。此外，不同的操作系统也以不同的方式处理它们。在本章中没有详细介绍它们的原因是：因为它们太多样化，并且不允许获得一致的读者体验（此外，也不是每一个人都具有游戏手柄）。

问：每个组件都可以通过脚本编辑吗?

答：是的，至少所有的内置组件都是如此。

9.7 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

9.7.1 问题

1．判断题：方法也可以称为函数。

2．判断题：并非所有的方法都具有返回类型。

3．把玩家交互映射到特定的按钮为什么是一件糟糕的事情？

4．在关于局部和目标组件的小节中的“TRY IT YOURSELF”练习中，沿着正x 轴平移立方体，并沿着z轴旋转它。这将导致立方体在一个大圆上四处移动，为什么？

9.7.2 答案

1．正确。

2．错误。每个方法都具有返回类型，如果方法不返回任何内容，其类型就是void。

3．如果玩家重新映射控制以满足他们的个人偏好，将会遇到更大的困难。通过把控制映射到一般的轴，玩家可以轻松地改变把哪些按钮映射到那些轴。

4．变换发生在局部坐标系统上（参见第2章）。因此，立方体确实沿着正x轴移动。不过，该轴相对于摄像机所面对的方向将持续改变。

9.7.3 练习

一种良好的做法是：把每一章的课程结合起来，查看它们如何以一

种更现实的方式交互。在这个练习中，你将编写一些脚本，允许玩家定向控制游戏对象。如果需要，在用于第9章（Hour 9）的本书配套资源中可以找到这个练习的解决方案。

1．创建一个新的项目或场景，向场景中添加一个立方体，并把它定位于(0, 0, –5)处。然后向场景中添加一个定向灯光。

2．创建一个名为Scripts的新文件夹，并且创建一个名为CubeControlScript的新脚本。然后把该脚本附加到立方体上。

尝试向脚本中添加以下功能。如果感到迷茫，可以检查用于第9章（Hour 9）的本书配套资源，以获取帮助。

无论何时玩家按下左、右箭头键，将分别沿着负x 轴和正x 轴移动立方体。无论何时玩家按下下、上箭头键，将分别沿着负y轴和正y轴移动立方体。

当玩家沿着y 轴移动鼠标时，就围绕x轴旋转立方体。当玩家沿着x轴移动鼠标时，就围绕y轴旋转立方体。

当玩家按下M 键时，就放大立方体；当玩家按下N键时，就缩小立方体。

第10章 碰撞

在本章中你将学到：

刚体的基础知识；

如何使用碰撞器；

如何利用触发器编写脚本；

如何投射光线。

在本章中，你将学习处理视频游戏中最常用的物理概念：碰撞（collision）。简单地讲，碰撞知道一个对象的边界何时接触到另一个对象。你首先将学习刚体是什么，以及它们可以为你做什么。之后，将通过把对象碰撞在一起，试验 Unity 强大的内置物理引擎。接着，你将学习利用触发器更精妙地使用碰撞。在本章最后，你将学习使用光线投射来检测碰撞。

10.1 刚体

为了让对象利用 Unity 内置的物理引擎，它们必须包括一个称为刚体（rigidbody）的组件。添加刚体组件将使对象的行为方式像真实的坚硬实体那样。要添加刚体组件，只需选择你想要的对象，并单击Component > Physics > Rigidbody命令即可。在Inspector 中将注意到给对象添加了新的刚体组件，如图10.1所示。

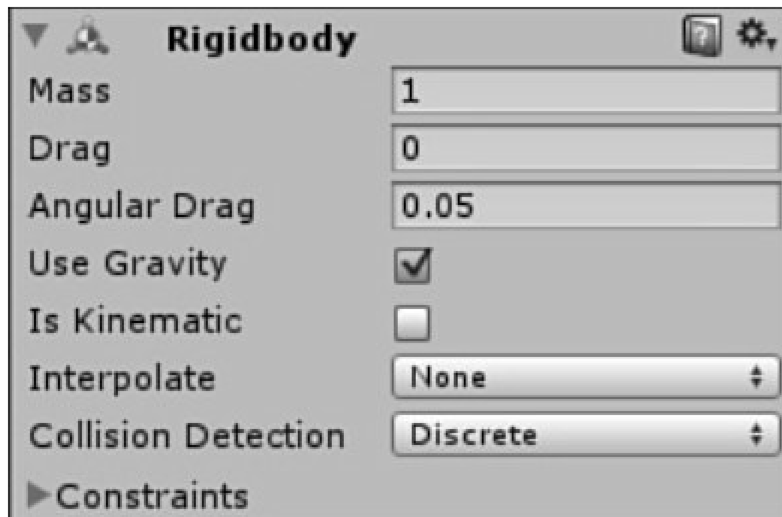


图10.1 刚体组件

刚体组件具有几个你还没有见过的新属性，表10.1描述了这些属性。

表10.1 刚体属性

属性	描述
Mass	任意单元中的对象的质量。较重的对象将具有较高的质量
Drag	对象在移动时应用于对象的空气阻力有多大。较高的阻力将使对象获得更大的力来移动，并将更快地阻止移动的对象。如果阻力为 0，则不会应用空气阻力
Angular Drag	非常像阻力，这是在转动时应用的空气阻力
Use Gravity	确定是否对这个对象应用 Unity 的重力计算。重力将或多或少地影响对象，这依赖于它的阻力
Is Kinematic	如果对象是运动学的，它将不会受 Unity 的物理学影响。当你想要一个刚体但是不希望使用 Unity 的老套物理学时，就可以使用这个属性
Interpolate	确定对象的运动多么以及是否平滑。默认情况下，它被设置为 Smooth。Interpolate 使平滑基于前一帧，而 Extrapolate 则基于下一个假定的帧。对于玩家对象，建议打开这个属性；对于其他的一切，则建议关闭它。这将给你提供最好的性能和质量
Collision Detection	确定碰撞是怎样计算的。Discrete 是默认设置，指示如何相对于彼此来测试所有的对象。如果在检测极快的对象的碰撞时遇到麻烦，Continuous 设置可能会有帮助。不过，要知道的是，Continuous 可能对性能具有较大的影响。Continuous Dynamic 设置对于其他的离散对象将使用离散检测，而对于其他的连续对象则使用连续检测
Constraints	约束是刚体施加于对象的运动限制。默认情况下，它们是关闭的。冻结位置轴将阻止对象沿着该轴移动，冻结旋转轴则将阻止对象围绕该轴旋转

使用刚体

让我们花一点时间看看刚体的实际应用。

（1）创建一个新的项目或场景，并场景中添加一个立方体，并把它置于(0, 1,−5)处。你可以向场景中添加一个定向灯光。

（2）运行场景，注意立方体怎样在摄像机前浮动。

（3）给对象添加一个刚体（单击Components > Physics > Rigidbody命令）。

（4）运行场景，注意对象现在怎样由于重力而下落。

（5）继续试验阻力和约束属性。

10.2 碰撞

既然你已经四处移动了对象，现在就应该开始使它们彼此碰撞。为了使对象检测碰撞，它们需要一个称为碰撞器（collider）的组件。碰撞器是投射在对象周围的边界，当其他对象进入其中时可以检测到它们。

注意：

碰撞的要求

值得一提的是，对象不需要刚体也能发生碰撞。碰撞只需所涉及的两个对象具有一个碰撞器对象即可。在本章中之所以包括了刚体，是因为它们通过支持对象下落而有助于说明主题。此外，刚体是触发碰撞所必需的，这将在本章后面介绍。

10.2.1 碰撞器

像球体、胶囊和立方体这样的几何对象在创建时就已经在它们上面具有碰撞器组件。对于没有碰撞器的对象，可以通过单击Component > Physics命令并从菜单中选择你想要的碰撞器形状，给对象添加碰撞器。图10.2显示了你可以选择的不同碰撞器形状。

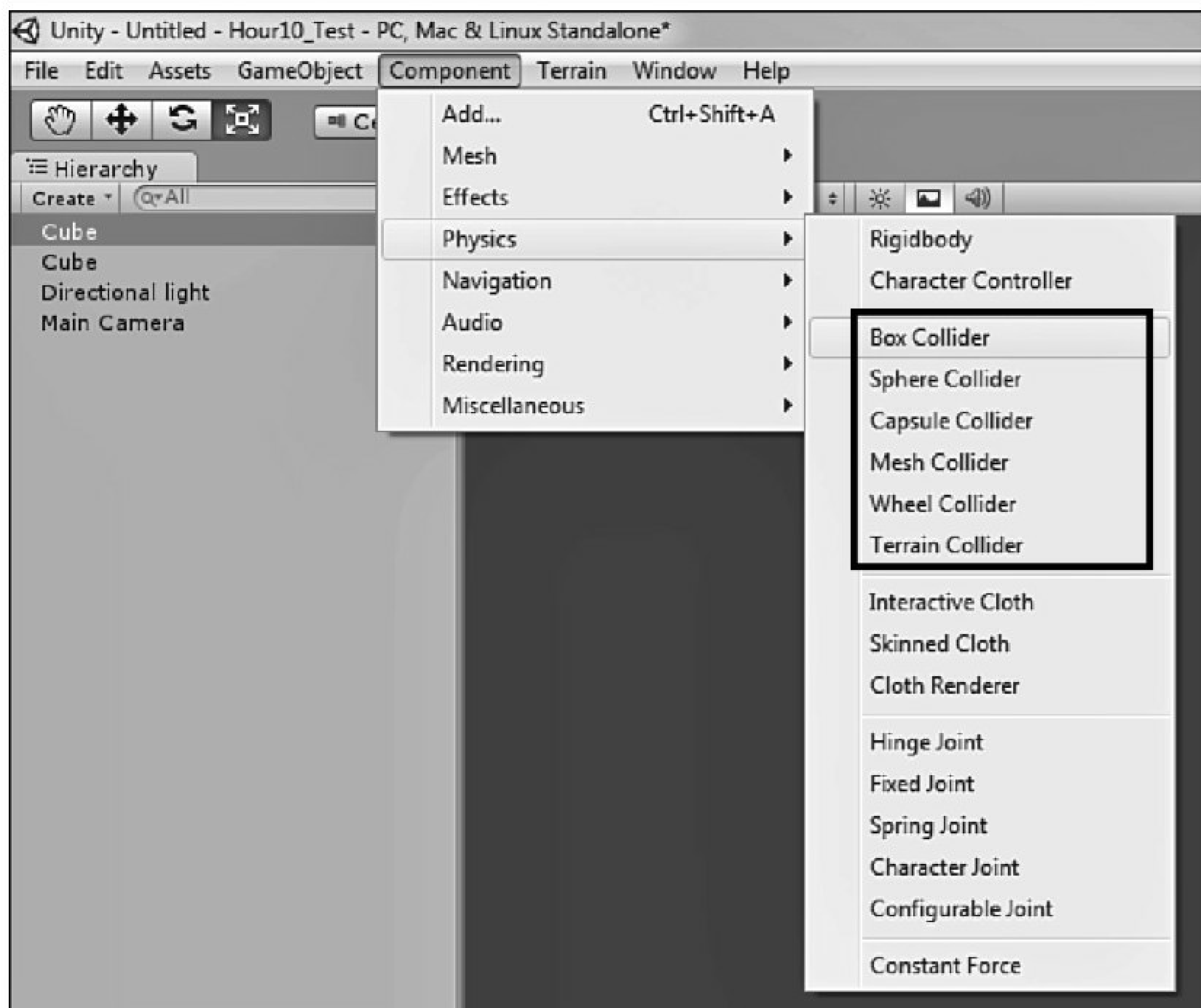


图10.2 不同的碰撞器

一旦给对象添加了碰撞器，碰撞器对象就会出现在Inspector中。表10.2描述了碰撞器属性。

表10.2 碰撞器属性

属性	描述
Is Trigger	确定碰撞器是物理碰撞器还是触发碰撞器。在本章后面将更详细地介绍触发器
Material	碰撞器使你能够对对象应用物理材质，以改变它们的行为方式。例如，可以使对象像木头、金属或橡胶那样。在本章后面将介绍物理材质
Center	碰撞器相对于包含对象的中心点
Size	碰撞器的大小
几何属性	如果碰撞器是一个球体或胶囊，就可能具有一个额外的属性，比如 Radius。它们的行为方式与你所期望的完全一样

提示：

混合与匹配碰撞器

在对象上使用不同形状的碰撞器可能会产生一些有趣的效果。例如，使立方体上的碰撞器比立方体大得多，将使得立方体看起来好像浮动在某个表面上一样。同样，较小的碰撞器将允许对象陷入表面中。此外，把球形碰撞器放在立方体上将允许立方体像球一样滚动。用不同的方式进行试验，使碰撞器适用于对象。

试验碰撞器

现在应该试验一些不同的碰撞器，看看它们如何交互。一定要保存这个练习，在本章后面将再次使用它。

（1）创建一个新的项目或场景，并向场景中添加两个立方体和一个定向灯光。把一个立方体放在(0, 1, −5)处，并在它上面放置一个刚体。把另一个立方体放在(0, −1, −5)处，并把它缩放到(4, .1, 4)，同时把旋转角度设置为(0, 0, 15)。在第二个立方体上也放置一个刚体，但是取消选中Use Gravity属性。

（2）运行场景，并且注意上面的立方体如何落到另一个立方体上。之后，它们都将下落并离开屏幕。现在，在下面的立方体上，在刚体组件的约束下，冻结位置和旋转的全部3根轴。

（3）运行场景，并且注意上面的立方体现在如何下落并停留在下面的立方体上。从上面的立方体删除盒碰撞器（右击Box Collider 组件，并选择Remove Component 命令）。给上面的立方体添加一个球形碰撞器（单击Component > Physics > Sphere Collider命令 ），并给下面的立方体提供(0, 0, 350)的旋转角度。

（4）运行场景，并且注意盒子像球体一样滚下斜坡，即使它是一个立方体。

（5）继续试验不同的碰撞器。另一个有趣的试验是更改下面的立方体上的约束，只尝试冻结y轴位置，并解除冻结所有其他的一切。试

试不同的方式，使盒子发生碰撞。

提示：

复杂的碰撞器

你可能注意到一个名为Mesh Collider 的碰撞器，在正文中专门略去了这个碰撞器，因为与其他碰撞器相比，它更多地是一种建模方面的实践。实质上，网格碰撞器（mesh collider）是一种具有 3D 模型的精确形状的碰撞器。这听上去是有用的，但是在实际中可能会极大地降低游戏的性能。此外， Unity对网格碰撞器中允许的多边形数量设置了严格的限制。要养成的更好的习惯，利用几个基本的碰撞器组成复杂的对象。如果具有一个人体模型，可以尝试用球体表示头部和手，以及用胶囊表示躯干、手臂和腿。这样可以节省性能，并且仍然具有非常敏锐的碰撞检测。

10.2.2 物理材质

可以对碰撞器应用物理材质，给对象提供变化的物理属性。例如，可以使用橡胶材质使对象具有弹性，或者使用冰材质使之变得滑溜。甚至可以创建你自己的材质，来模拟你所选的特定材质。

要导入 Unity 使之可用的材质，可以单击 Assets > Import Package > Physic Materials命令。在Import屏幕中，保持选择所有的选项，并单击Import按钮。这将把Standard Assets文件夹引入到项目中，其中包含用于Bouncy、Ice、Metal、Rubber和Wood的材质。要创建一种新的物理材质，可以在Project视图中右击Assets文件夹，并选择Create > Physic Material命令。

物理材质具有一组属性，它们确定了材质在物理层面的行为方式，如图 10.3 所示。表10.3描述了物理材质的属性。要把一种物理材质应用于某个对象，只需在Project视图中把它拖到该对象上即可。

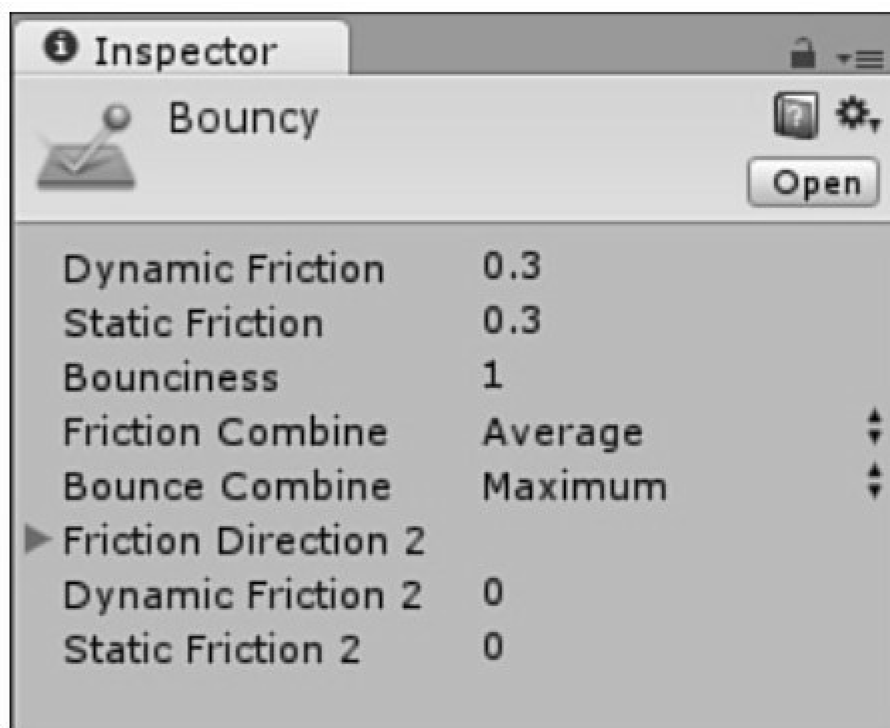


图10.3 物理材质的属性

表10.3 物理材质的属性

属性	描述
Dynamic Friction	当对象已经在移动时应用的摩擦力。较低的数字表示对象更光滑
Static Friction	当对象处于静止状态时应用的摩擦力。较低的数字表示对象更光滑
Bounciness	通过碰撞保留的能量值。值 1 将导致对象弹跳，而不会损失任何能量；它将永远弹跳。值 0 将阻止对象弹跳
Friction Combine	确定如何计算两个碰撞对象的摩擦力。可以计算平均的摩擦力，也可以使用最小或最大摩擦力，或者可以把它们相乘起来
Bounce Combine	确定如何计算两个碰撞对象的弹力。可以计算平均的弹力，也可以使用最小或最大弹力，或者可以把它们相乘起来
Friction Direction 2	如果希望对象在特定的方向具有不同的摩擦力，就可以指定这个属性（可以考虑滑冰）
Dynamic Friction 2	像 Dynamic Friction 一样，只应用于指定的方向。仅当提供了 Friction Direction 2 时才能够使用它
Static Friction 2	像 Static Friction 一样，只应用于指定的方向。仅当提供了 Friction Direction 2 时才能够使用它

物理材质的效果可以像你想象的那样微妙或显著。可以自己试试它，看看你可以创建哪些有趣的行为。

10.3 触发器

迄今为止，你已经见过了物理碰撞器，它们是使用 Unity 的内置物理引擎以位移和旋转方式做出反应的碰撞器。不过，如果回想一下第7章“第1 款游戏：Amazing Racer”，就可能会记得使用另一种类型的碰撞器。还记得当玩家进入水障和完成区域时是如何检测游戏的吗？这是触发碰撞器在工作。触发器可以检测碰撞，就像正常的碰撞器那样，但它不会做任何特定于碰撞的事情。触发器将代之以调用3个特定的方法，允许你（即程序员）确定碰撞意味着什么：

```
void OnTriggerEnter(Collider other)//is called when an object enters the trigger
void OnTriggerStay(Collider other)//is called when an object stays in the trigger
void OnTriggerExit(Collider other)//is called when an object exits the trigger
```

使用这些方法，无论何时对象进入碰撞器、停留在碰撞器中或者离开碰撞器，都可以定义所发生的事情。例如，无论何时一个对象进入立方体的周界，都希望把一条消息写到控制台，那么就可以给立方体添加一个触发器，然后利用以下代码把一个脚本附加到立方体上：

```
void OnTriggerEnter(Collider other)
{
    print("Object has entered collider");
}
```

注意：

触发器不工作

为了让触发碰撞器工作，必须涉及到一个刚体。如果不带有刚体的对象进入触发碰撞器，那么什么也不会发生。如果在屏幕中注意到一些对象没有像你想要的那样触发，就要确保它们上面具有刚体。

你可能注意到触发器方法的一个参数，即碰撞器类型的变量other。这是一个指向进入触发器的对象的引用。使用该变量，可以用你想要的任何方式操纵对象。例如，如果希望修改上面的代码，把进入触发器的对象的名称写到控制台，可以编写如下代码：

```
void OnTriggerEnter(Collider other)
{
   print(other.gameObject.name + " has entered the trigger");
}
```

甚至可以更进一步，利用以下代码销毁进入触发器的对象：

```
void OnTriggerEnter(Collider other)
{
   Destroy(other.gameObject);
}
```

使用触发器

在这个练习中，你将有机会利用一个正常工作的触发器构建交互式场景。在用于第10章（Hour　10）的本书配套资源中可以找到用于这个练习的完成的项目，它的名称是Hour10_TriggerExercise。

（1）创建一个新的项目或场景。向场景中添加一个定向灯光，然后添加一个立方体和一个球体，并把立方体和球体分别放置在(−1, 1, −5)和(1, 1, −5)处。

（2）创建两个名为TriggerScript和MovementScript的脚本，然后把触发器脚本放在立方体上，并把移动脚本放在球体上。

（3）在立方体的碰撞器上，选中IsTrigger。给球体添加一个刚体，并取消选中Use Gravity。

（4）向移动脚本的Update方法中添加以下代码：

```
float mX = Input.GetAxis("Mouse X")/ 10;
float mY = Input.GetAxis("Mouse Y")/ 10;
transform.Translate(mX, mY, 0);
```

（5）向触发器脚本中添加以下代码，确保把这些代码放在任何方法的外面，但是要放在类里面：

```
void OnTriggerEnter(Collider other)
{
   print(other.gameObject.name + " has entered the cube");
}
void OnTriggerStay(Collider other)
{
   print(other.gameObject.name + " is still in the cube");
}
void OnTriggerExit(Collider other)
{
   print(other.gameObject.name + " has left the cube");
}
```

（6）运行场景，注意鼠标怎样移动球体。利用立方体碰撞球体，并且注意控制台输出。注意两个对象没有物理地做出反应，但是它们仍然在交互。

10.4 光线投射

光线投射（raycasting）是发出一根假想的线（即光线）并观察它撞上某样物体之后的动作。例如，想象一下通过望远镜观察物体。你的视线就是光线，你在另一端看到的任何内容就是光线所撞上的物体。游戏开发人员随时都会使用光线投射，用于像瞄准、确定视线、测量距离等之类的动作。Unity中有几个Raycast方法，这里将展示两个最常用的方法。第一个Raycast方法如下所示：

bool Raycast(Vector3 origin, Vector3 direction, float distance, LayerMask mask）;

注意：

这个方法接受相当多的参数。请注意，它使用一个名为 Vector3 的变量。Vector3 是一个在其内保存3个浮点数的变量类型，它是一种无需3个单独的参数即可指定x、y和z坐标的极佳方式。第一个参数origin是光线开始的位置；第二个参数direction是光线行进的方向；第三个参数float确定光线行进的距离；最后一个变量 mask 则确定光线将会撞上哪些层。可以省略distance和mask变量，如果这样做，光线将会行进无限远的距离，并且会撞上所有的对象类型。

如前所述，利用光线可以做许多事情。例如，如果想要确定摄像机前是否有某个物体，可以利用以下代码附加一个脚本：

```
void Update(){
   //cast the ray from the camera's position in the forward direction
   if (Physics.Raycast(transform.position, transform.forward, 10))
      print("There is something in front of the camera!");
}
```

可以使用这种方法的另一个方式是查找光线将会碰撞的对象。该方法的这个版本使用一个名为RaycastHit的特殊变量类型。Raycast方法的许多版本都会利用（或者不利用）距离和层屏蔽。不过，使用该方法的这个版本的最基本的方式看起来如下：

```
bool Raycast(Vector3 origin, Vector3 direction, out Raycast hit, float distance);
```

关于该方法的这个版本有一件新的、有趣的事情。你可能注意到它使用了一个以前未见过的新关键字：out。这个关键字意味着：当方法在运行时，变量hit将包含被撞上的任何对象。该方法在执行完成时，实际上将发回值。

投射一些光线

让我们创建一个交互式的“射击”程序。这个程序将从摄像机发送一根光线，并将销毁它接触到的任何对象。在用于第10章（Hour 10）的本书配套资源中可以找到用于这个练习的完成的项目，其名称为Hour10_RaycastExercise。

（1）创建一个新的项目或场景。向场景中添加4个球体，并把它们的名称改为Sphere1～Sphere4。然后把这些球体分别放在(−1,1,−5)、(1, 1.5, −5)、(−1,−2,5)和(1.5,0,0)处。

（2）创建一个名为RaycastScript 的新脚本，并把它附加到Main Camera 上。在该脚本的Update方法内，添加以下代码：

```
float dirX = Input.GetAxis("Mouse X");
float dirY = Input.GetAxis("Mouse Y");
//opposite because we rotate about those axes
transform.Rotate(dirY, -dirX, 0);
CheckForRaycastHit(); //this will be added in the next step
```

（3）现在，通过在方法外面但是在类里面添加以下代码，向脚本中添加CheckForRaycastHit()方法：

```
void CheckForRaycastHit()
{
    RaycastHit hit;
    if(Physics.Raycast(transform.position, transform.forward, out hit))
    {
        print (hit.collider.gameObject.name + " destroyed!");
        Destroy(hit.collider.gameObject);
    }
}
```

（4）运行场景，注意移动鼠标将怎样移动摄像机。尝试把摄像机放在每个球体的中心，注意球体是怎样销毁的，并且把消息写到控制台。

10.5 小结

在本章中，你学习了通过碰撞实现对象交互，还学习了 Unity 利用刚体实现物理能力的基础知识。然后，你接触了多种类型的碰撞器和碰撞。接着，你认识到碰撞不仅仅只是当接触到触发器时物体会反弹。最后，你学习了通过光线投射来查找对象。

10.6 问与答

问：我的所有对象都应该具有刚体吗？

答：刚体是主要充当物理角色的有用的组件。也就是说，给每个对象添加刚体可能具有怪异的副作用，并且可能会降低性能。一条很好的经验法则是：仅当需要组件时才添加它们，而不要抢先添加。

问：有多种碰撞器我们还没有讨论，为什么不讨论它们？

答：大多数碰撞器的行为方式与我们介绍过的一些碰撞器相同，或者超出了本书的范围。因此，本书省略了它们。我只想说，本书仍然提供了你制作一些非常有趣的游戏所需要的知识。

10.7 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

10.7.1 问题

1．如果你希望对象展示像下落这样的物理轨迹，则对象上将需要这个组件。

2．判断题：对象上面只能有一个碰撞器。

3．判断题：为了使触发器工作，触发器对象也需要一个刚体。

4．光线投射有什么用途？

10.7.2 答案

1．刚体。

2．错误。一个对象上面可以具有许多不同的碰撞器。

3．错误。带有触发器的任何碰撞器都需要具有刚体。

4．用于确定一个对象可以看到什么并且在途中查找对象，以及找到对象之间的距离。

10.7.3 练习

在这个练习中，你将创建一个利用运动和触发器的交互式应用程序。该练习需要你创造性地确定一种解决方案（因为这里没有展示它）。如果你感到迷惑并且需要帮助，可以在用于第10章（Hour 10）的本书配套资源中找到这个练习的解决方案，其名称是Hour10_Exercise。

1．创建一个新的项目或场景。向场景中添加一个定向灯光，然后向场景中添加一个立方体，并把它定位于(−1.5, 0, −5)处。把立方体缩放为(.1, 2, 2)，并把它重命名为LTrigger。

2．复制立方体（在Hierarchy视图中右键单击立方体，并选择Duplicate命令）。把新立方体命名为RTrigger，并把它定位于(1.5, 0, −5)处。

3．向场景中添加一个球体，并把它定位于(0, 0, −5)处。然后给球体添加一个刚体，并取消选中Use Gravity。

4．创建一个名为TriggerScript的脚本，并把它同时放置在LTrigger和RTrigger上。然后创建一个名为MotionScript的脚本，并把它放置在球体上。

现在处理有趣的部分。你将需要在应用程序中创建以下功能。

玩家应该能够利用箭头键移动球体。

当球体进入、离开或停留在任何一个触发器内时，应该把相应的消息写到控制台上。

此外，还应该把球体进入的触发器的名称与上述消息一起写到控制台上。

第11章 第2款游戏：Chaos Ball

在本章中你将学到：

怎样设计Chaos Ball 游戏；

怎样构建ChaosBall 舞台；

怎样构建ChaosBall 实体；

怎样构建ChaosBall 控制对象；

怎样进一步改进ChaosBall。

现在应该再次利用你所学的知识制作另一款游戏。在本章中，你将制作 Chaos Ball 游戏，它是一款快节奏的街机风格的游戏。本章首先将介绍游戏的基本设计元素。接着，将构建舞台和游戏对象。每个对象都将被创建为独特的对象，并为其提供特殊的碰撞属性。然后，将添加交互性，使玩家能够玩游戏。最后将尝试玩游戏并执行任何必要的调整，以改进游戏体验。

提示：

完成的项目

一定要遵循本章中的指导，构建完整的游戏项目。万一你感到困惑，可以在用于第11章（Hour 11）的本书配套资源中找到该游戏的完成的副本。看看你是否需要帮助或灵感！

11.1 设计

在第7章中，你已经学习过设计元素是什么。这一次，你将开始接触到它们。

11.1.1 理念

这款游戏稍微有点像Pinball或Breakout。玩家将身处舞台中。4个角中的每个角都将具有一种颜色，而具有对应颜色的4只球将四处浮动。在4只彩球中间，还将有多只黄球，称为混乱球（chaos ball）。这些球只是为了挡住你的路，并使游戏具有挑战性。它们比4只彩球要小一些，但是它们也移动得更快。玩家将看见一个平坦的表面，他们将尝试利用它把彩球撞进正确的角落。

11.1.2 规则

游戏规则将说明如何玩游戏，同时还将间接提及对象的一些属性。用于Chaos Ball 游戏的规则如下。

当全部4 只球都位于正确的角落时，玩家将获胜。不存在输掉游戏的情况。、

击中正确的角落将导致一只球变成不活动状态。

游戏中的所有对象都具有超级弹性（它们不会因为碰撞而损失动量）。

任何球（或玩家）都不能离开舞台。

彩球的速度和混乱球的速度是随机的。

11.1.3 需求

这款游戏的需求很简单。它不是一款图形密集的游戏，而是以依靠脚本和交互来实现其娱乐效果。Chaos Ball 游戏的需求如下。

地形的墙壁，用于充当舞台。

用于地形和游戏对象的纹理，它们是在Unity标准资源中提供的。

多只彩球和混乱球，将在Unity中生成它们。

一个角色控制器，它是由Unity标准资源提供的。

一个游戏控制器，将在Unity中创建它。

一种具有弹性的物理材质，将在Unity中创建它。

彩色的角落指示器，将在Unity中生成它们。

交互式脚本，将在MonoDevelop 中编写它们。

11.2 舞台

你首先想要创建的是用于让动作发生的舞台。选择舞台（arena）这个术语是为了提供下面这种思想：即地形相当小，并且四面是围住的。玩家和任何球都应该不能离开舞台。除此之外，舞台将相当简单，如图11.1所示。

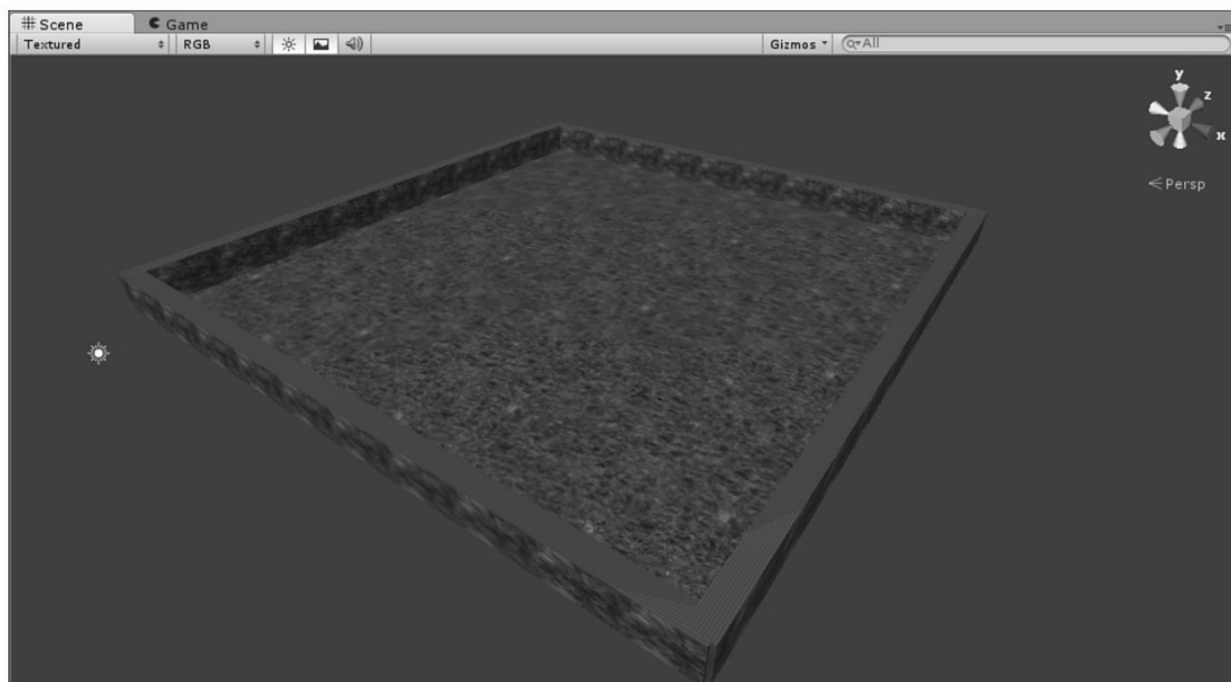


图11.1 舞台

11.2.1 创建舞台

如前所述，由于基本的舞台地图比较简单，这将是一个简单的过程。要创建舞台，可以遵循下面这些步骤。

（1）在名为ChaosBall 的文件夹中创建一个新项目。这一次将在Create New Project 对话框中，选中Character Controller.unityPackage和Terrain Assets.unityPackage 旁边的复选框，如图11.2所示。然后向项目

中添加一种地形。

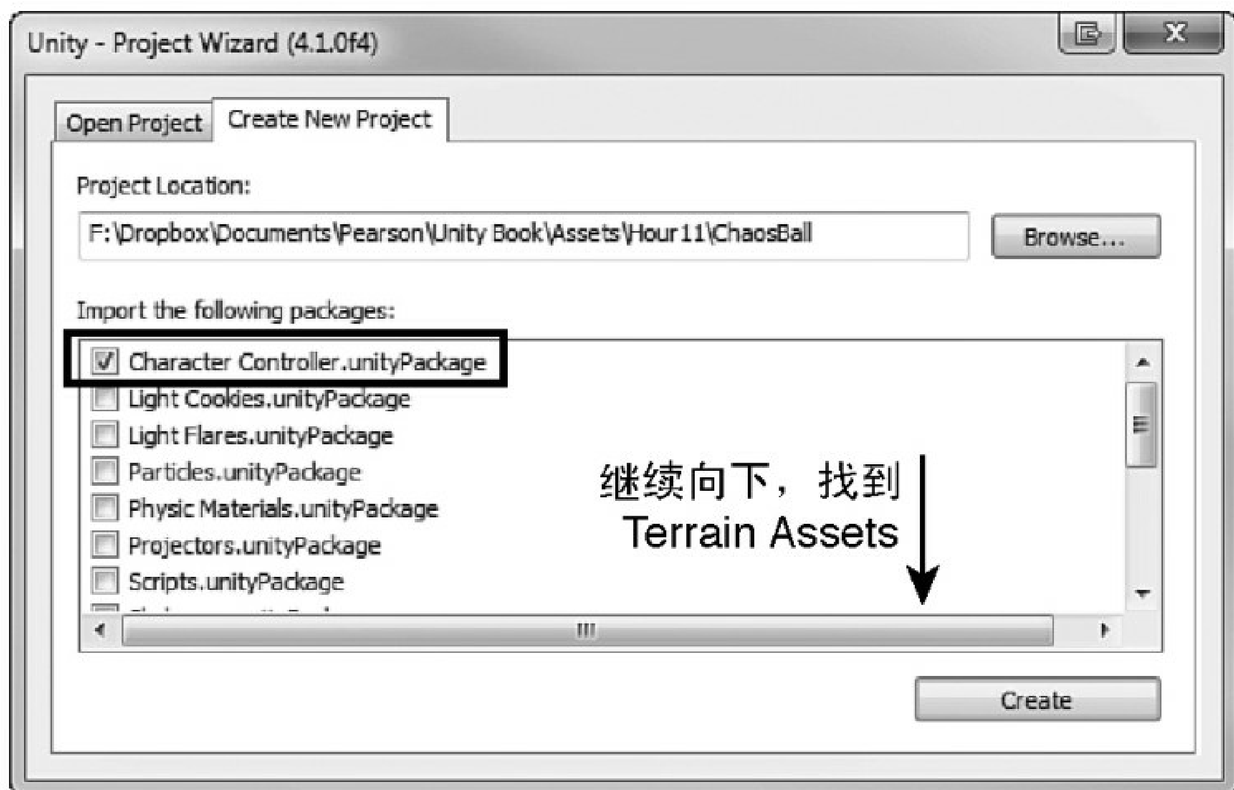


图11.2 Create New Project对话框

（2）把地形的分辨率设置为50×50（记住，它是在Terrain Settings的Resolution 区域中设置的）。向场景中添加一个定向灯光，并删除Main Camera。

（3）向场景中添加一个立方体，把该立方体置于(0, 1.5, 25)处，并把它缩放为(1.5, 3, 51)。注意它如何变成舞台的侧壁。然后把立方体重命名为Wall。

（4）在Scenes文件夹中把场景重命名为Main。

提示：

合并对象

你可能感到迷惑的是，舞台需要4面墙，可你只创建了一面。其思想是：你希望尽可能少地做冗余、乏味的工作。通常，如果需要多个非常相似的对象，就可以只创建一个对象，然后把它复制多次。在这种情

况下，可以利用合适的材质和属性建立一面墙，然后简单地把它复制3次。可以为角落节点、混乱球和彩球重复相同的过程。顺利的话，可以看到只需进行一点规划就可以节省相当多的时间。

11.2.2 纹理化

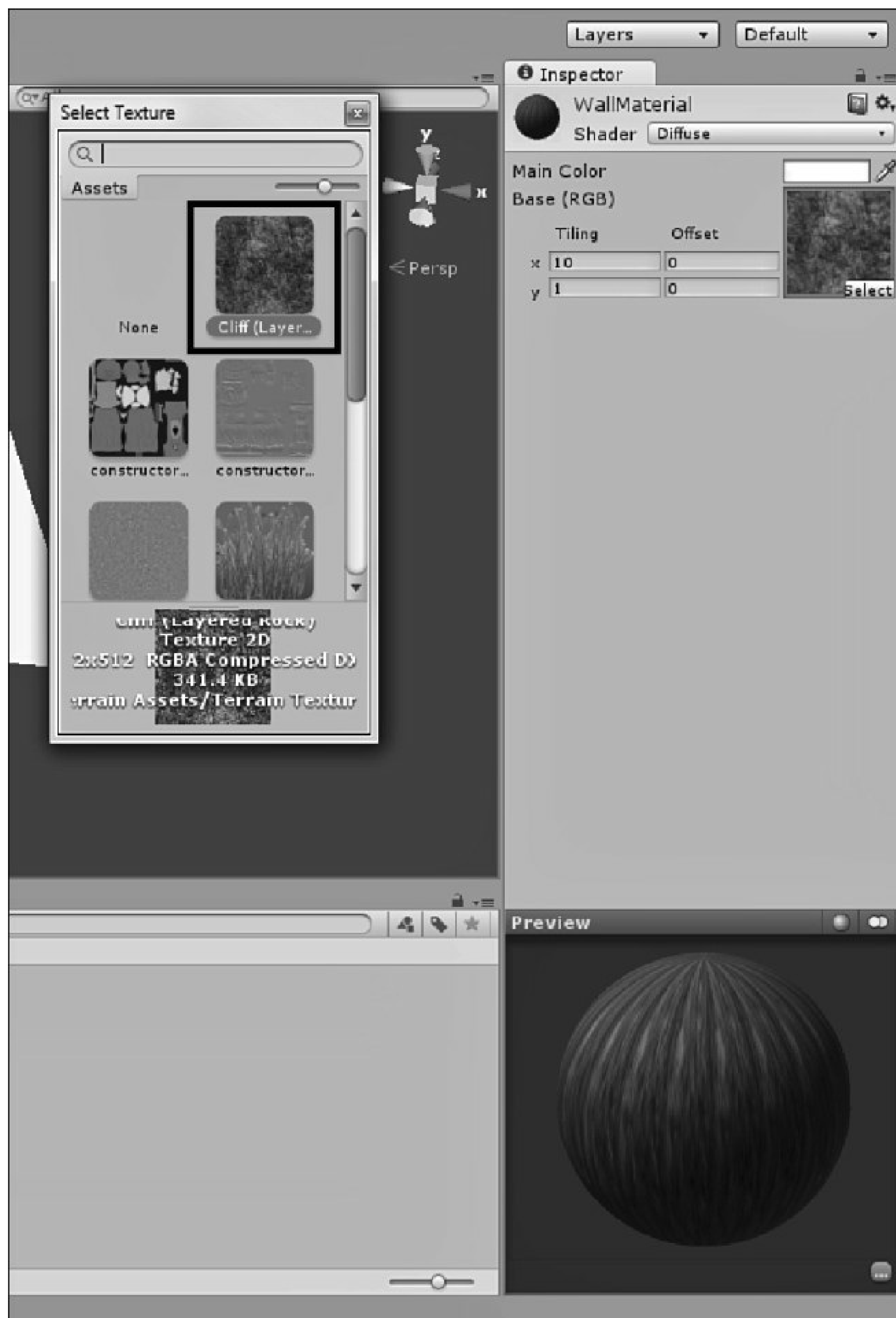
目前，舞台看起来非常简陋和平淡。一切都是白色的，并且只有一面墙。下一步是添加一些纹理，使舞台显得有生气。你需要专门纹理化两个对象：墙壁和地面。在完成这个步骤的过程中，可以自由地试验纹理化。如果喜欢，可以使之变得更有趣！

（1）在Project视图中的Assets下面创建一个名为Materials的新文件夹。向该文件夹中添加一种材质（右键单击文件夹，并选择Create > Material命令），并把该材质命名为WallMaterial。

（2）把x轴贴图设置为10，如图11.3所示。

（3）在Inspector视图中对墙壁材质应用Cliff (Layered Rock)纹理，如图11.3 所示。

（4）在Scene视图中单击墙壁材质，并把它拖到墙壁对象上。



Layers
Default
Inspector
WallMaterial
Shader
Diffuse
Main Color
Base (RGB)
Tiling
Offset
x
10
0
y
1
0
Select
Select Texture
Assets
None
Cliff (Layer...
constructor...
constructor...
Texture 2D
341.4 KB
y
x
Persp
Preview

图11.3 给材质添加峭壁纹理

接下来，将需要纹理化地面。回想一下，由于地面是一种地形，它的纹理化将稍有不同。

（1）选取地形，然后在Inspector视图中选择地形纹理化工具，如图11.4所示。

（2）单击Edit Textures > Add Texture命令，在Add Terrain Texture对话框中，选择Grass (Hill)纹理，并单击Add按钮。

（3）现在将用青草纹理化地形。

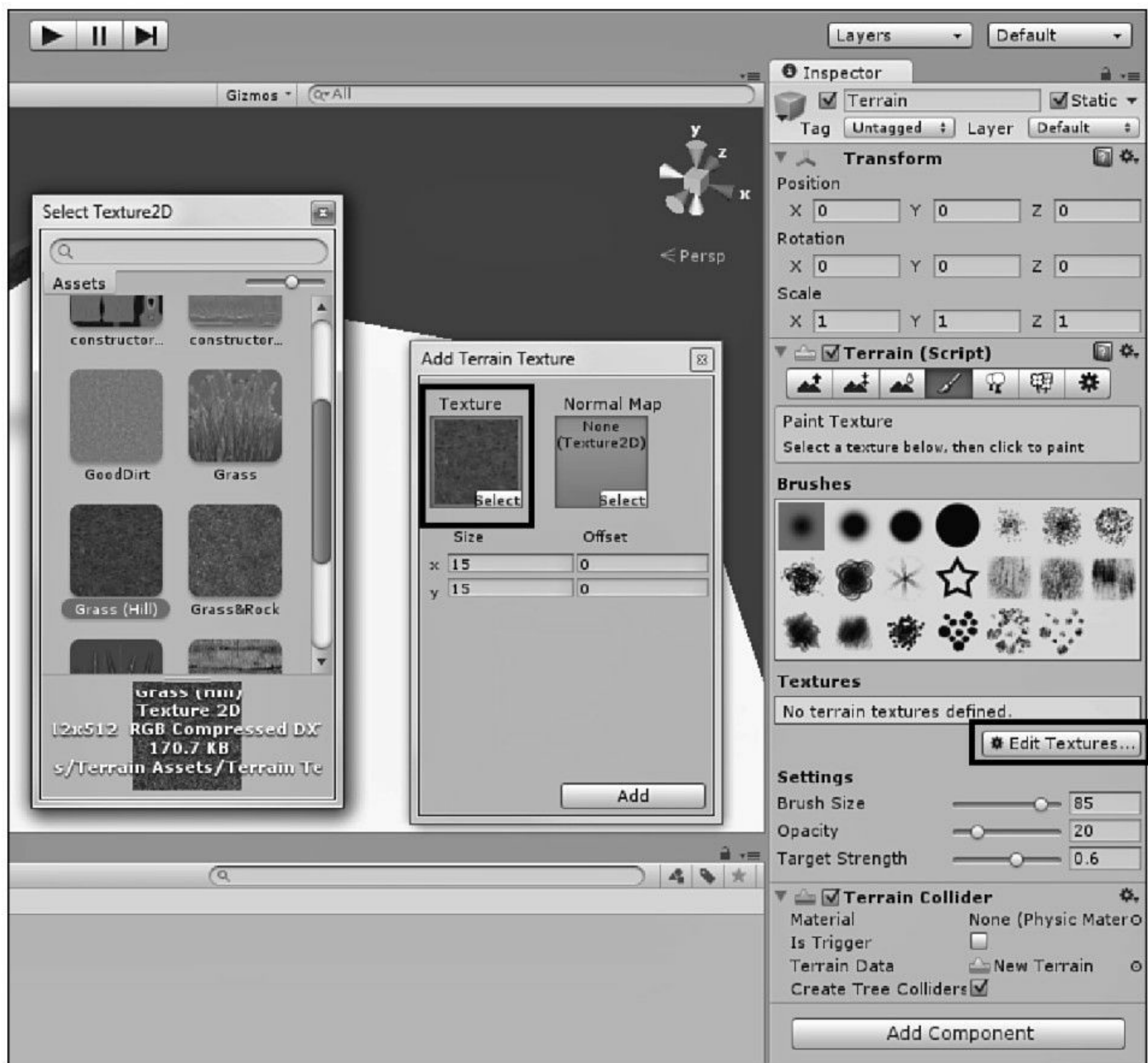


图11.4 添加地形纹理

11.2.3 超级弹性材质

你希望对象在弹离墙壁时不会损失任何动量，因此所需要的是一种超级弹性材质。回忆可知，Unity 具有一组物理材质可用。不过，它们提供的弹性材质并也不能满足你所需要的弹性。因此，你需要创建一种新材质，如下所示。

（1）右键单击 Materials 文件夹，并选择 Create > Physic Material

命令。把材质命名为SuperBouncyMaterial。

（2）如图11.5所示，设置超级弹性材质的属性。实质上，你希望把一切会减少能量的物体减至最少。

（3）单击超级弹性材质，并把它拖到场景中的墙壁对象上。它将自动作为物理材质应用于碰撞器。你应该会看到该材质列出在Box Collider 组件的Material 属性中。

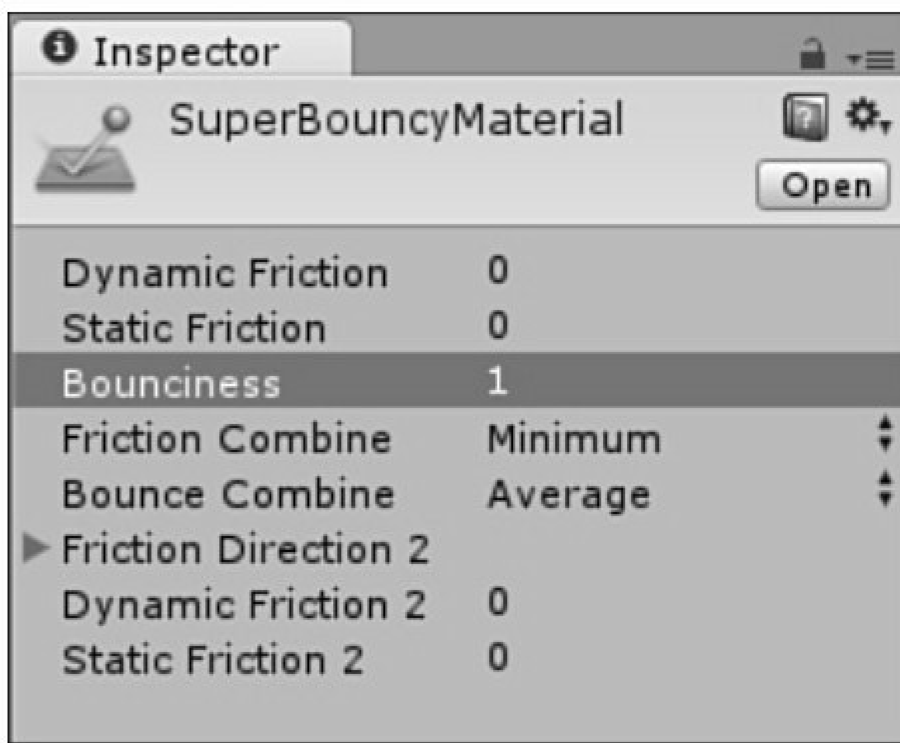


图11.5 SuperBouncyMaterial设置

11.2.4 完成舞台

既然墙壁和地面都已经完成了，现在就可以完成舞台。困难的工作已经做完了，现在只需复制墙壁（在Hierarchy视图中单击右键，并选择Duplicate命令）。准确的步骤如下。

（1）把墙壁复制一次，并把新实例放在(50, 1.5, 25)处。

（2）把墙壁再复制一次，把它放置在(25, 1.5, 0)处，并把旋转角度

设置为(0, 90, 0)。

（3）复制上一步中创建的墙壁（旋转过的墙壁），并把它放置在(25, 1.5, 50)处。

你的舞台现在应该具有4面墙壁，并且没有任何间隙或接缝（参见图11.1）。

11.3 游戏实体

在本节中，你将创建玩游戏所需要的多个游戏对象。就像舞台墙壁一样，创建每个实例的一个实例然后复制它将更容易。

11.3.1 玩家

这款游戏中的玩家将是一个改进的 First Person（“第一人称”）角色控制器。在创建这个项目时，你应该选择以导入那个角色控制器的程序包。继续前进，然后单击并把一个 First Person 角色控制器拖到场景中。把该控制器放在(46, 1, 4)处，并把旋转角度设置为(0, 315, 0)。

你想要做的第一件事是上移摄像机，使之离开控制器。这将使玩家在玩游戏时具有更大的视野。为此，可遵循下面这些步骤。

（1）在 Hierarchy 视图中展开 First Person 控制器（单击其名称旁边的箭头），并且定位Main Camera。你知道自己具有正确的Main Camera，因为它将是蓝色的。

（2）在选择了控制器的摄像机之后，把它定位于(0, 5, −3.5)处，并把旋转角度设置为(43, 0, 0)。摄像机现在应该位于控制器的上方和后面，并且稍微有点俯视控制器。

下一件要做的事情是给控制器添加一个减震器。减震器将是一个平坦的表面，玩家将在它上面弹球。为此，可以遵循下面这些步骤。

（1）向场景中添加一个立方体，并把立方体重命名为Bumper，然后把它缩放为(3.5, 3, 1)。

（2）单击超级弹性材质，并把它拖到减震器上。

（3）在Hierarchy视图，单击减震器并把它拖到First Person 控制器

上。这将把减震器嵌套在控制器上。然后，把减震器的位置改为(0, 0, .1)，并把旋转角度设置为(0, 0, 0)。减震器现在将稍稍位于控制器的前面。

最后一件要做的事情是把玩家的速度提高一点。选择First Person 控制器，并在Inspector视图中展开Character Motor (Script)组件的Movement属性，然后把最大前进速度改为11，并把最大横向速度改为10。

11.3.2 混乱球

混乱球将是快速、胡乱滚动的球，它们将在舞台上四处飞动，并且会干扰玩家。在许多方面，它们都类似于彩球，因此将给它们提供普遍适用的资源。要创建第一只混乱球，可以遵循下面这些步骤。

（1）向场景中添加一个球体。把球体重命名为Chaos，然后把它定位于(15, 2, 25)处，并将其缩放为(.5, .5, .5)。

（2）单击并把超级弹性材质拖到球体上。

（3）为混乱球创建一种新材质（非物理材质），并把它命名为ChaosBallMaterial。在材质的颜色选择器中，选择一种亮黄色，如图11.6所示。单击并把材质拖到球体上。

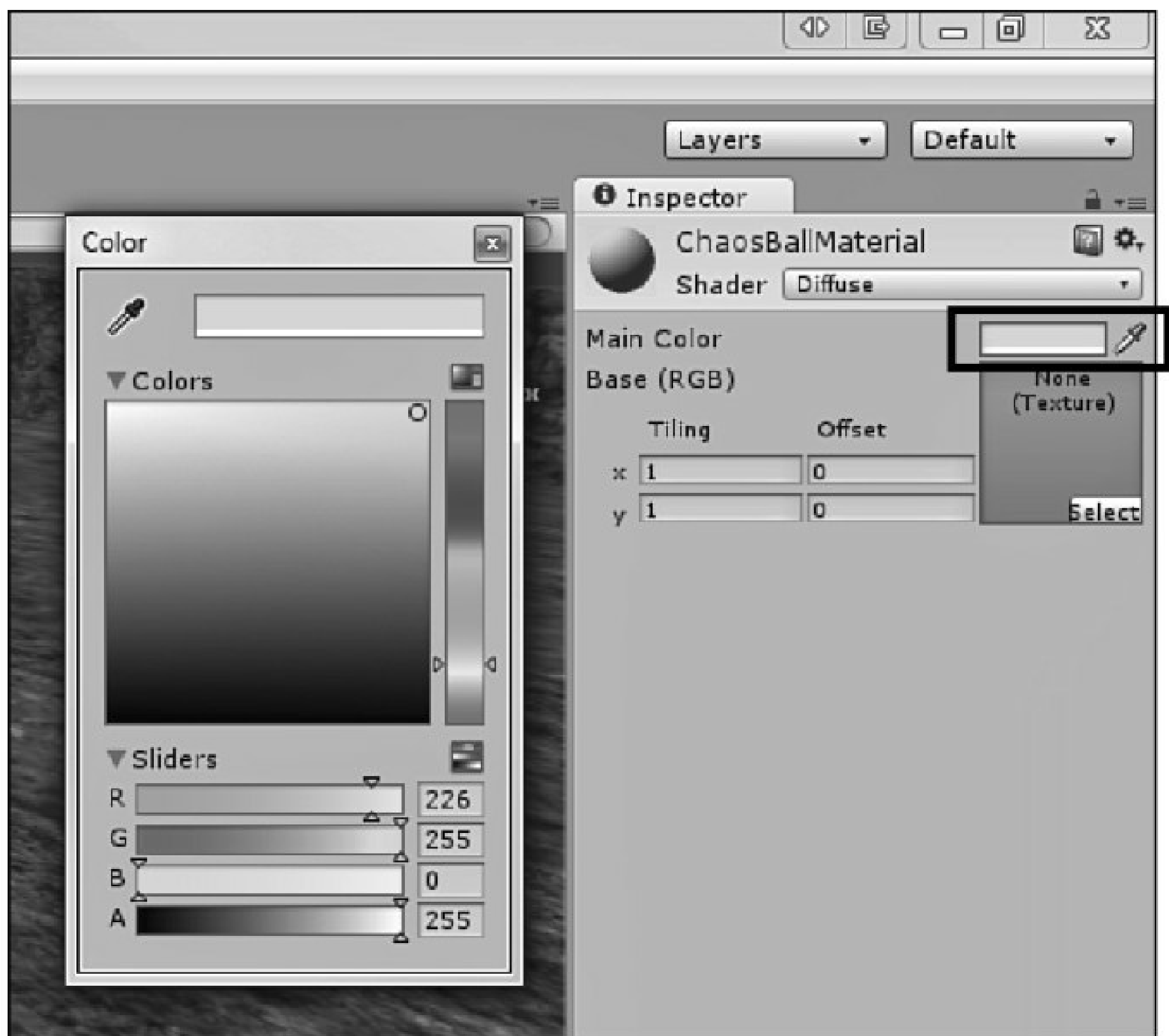


图11.6 ChaosBallMaterial设置

（4）给球体添加一个刚体。把角阻力改为 0，并取消选中 Use Gravity。把 Collision Detection属性改为Continuous。在Constraints属性下，冻结y位置。我们不希望球能够上下运动。

（5）打开Tag Manager（单击Edit > Project Settings > Tags 命令），通过单击Tags 旁边的箭头展开Tags 区域，并在Element 0处添加标签Chaos。然后继续前进，并添加标签Green、Orange、Red和Blue，以后将用到它们。

（6）选取混乱球，然后在Inspector视图中把它的标签改为Chaos，

如图11.7所示。

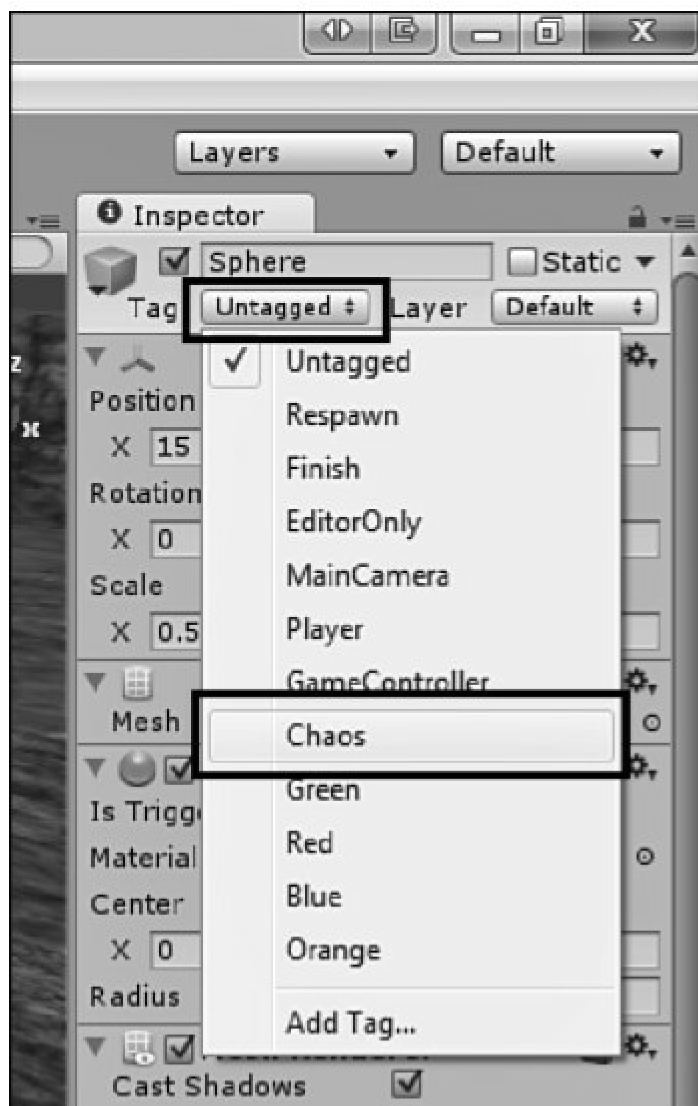


图11.7 选择Chaos标签

球现在就完成了，但它还不会做任何事情。你需要创建一个脚本，在舞台上四处移动球。你需要创建一个名为VelocityScript的脚本，并把它附加到混乱球上。程序清单11.1包含了用于速度脚本的完整代码。

程序清单11.1 VelocityScript.cs

```
using UnityEngine;
using System.Collections;
public class VelocityScript : MonoBehaviour {
```

```
    public float max = 50;
    // Use this for initialization
    void Start () {
        rigidbody.velocity = new Vector3(Random.Range(0, max), 0,
        Random.Range(0, max));
    }
    // Update is called once per frame
    void Update () {
    }
}
```

在这个程序清单中可以看到，使用Random.Range()方法给球提供一个沿着x轴和z轴的0～50之间的初始速度。Random.Range()方法接受两个数字作为参数，并返回它们之间的一个随机数字。

运行场景，并且观察球开始在舞台上飞来飞去。此时，混乱球就完成了。在Hierarchy视图中，把混乱球复制4次。把每只球散布在舞台周围（确保只改变x和z位置），并给它们都提供一个随机的y轴旋转角度。记住，沿着y轴的移动被锁定，以确保每只球都停留在y等于2的位置。

11.3.3 彩球

虽然混乱球是黄色的，并且它是一种颜色，但是彩球是赢得游戏所需要的4只特定的球。它们将是红色、橙色、蓝色和绿色。与混乱球一样，可以先制作一只球，然后复制它，从而使创建过程变得更容易。

要创建第一只球，可以遵循下面这些步骤。

（1）向场景中添加一个球体，并把它重命名为Blue。把该球体定位在靠近舞台中间的某个位置，并且确保它的y轴位置是2。

（2）创建一种名为 BlueMaterial 的新材质，并把它的颜色设置为蓝色，就像你对混乱球所做的那样，如图11.6所示。然后，继续创建RedMaterial、GreenMaterial和OrangeMaterial这些材质，并把它们设置为合适的颜色。单击并把BlueMaterial材质拖到球体上。

（3）单击并把超级弹性材质拖到球上。

（4）给球体添加一个刚体。然后在 Constraints 下把它的角阻力改为 0，取消选中 Use Gravity，并冻结y位置。

（5）你在前面创建了 Blue 标签。现在，把球体的标签改为 Blue，就像对混乱球所做的那样，如图11.7所示。

（6）把速度脚本附加到球体上。在Inspector 视图中，定位Velocity Script (Script)组件，并把Max属性改为25，如图11.8所示。这将导致球体最初比混乱球移动得慢一些。



图11.8 更改Max属性

如果现在运行场景，应该会看到蓝色的球在舞台上快速移动。现在需要创建另外3只球，它们都将是蓝色的球的复制品。要创建其他几只球，可以遵循下面这些步骤。

（1）复制蓝色的球，并用它们的颜色重命名新球：Red、Orange和Green。

（2）给新球提供一个与其名称对应的标签，使名称与标签具有相同的含义很重要。

（3）把合适的彩色材质拖到新球上，使球具有与其名称相同的颜色很重要。

（4）在舞台中给球提供一个随机的位置和旋转角度，但是要确保它的y轴位置是2。

此时，游戏实体就完成了。如果运行场景，将会看到所有的球都会在舞台上弹跳。

11.4 控制对象

既然一切都准备就绪，现在就应该游戏化它们。也就是说，现在应该把它们转变成可以玩的游戏。为此，需要创建4个角的球门、球门脚本和游戏控制器。一旦完成，你自己就拥有了一款游戏。

11.4.1 球门

舞台上的4个角中的每个角都具有与彩球对应的特定彩色球门。球门背后的思想是：当球进入球门时，球门将检查它的标签。如果标签与球门的颜色匹配，它就是匹配的。如果找到一个匹配，就把球设置为Kinematic（记住这使之处于不活动状态），并把球门设置为Solved。与以前的球对象一样，可以配置单个球门，然后复制它以匹配你的需要。

要建立初始的球门，可以遵循下面这些步骤。

（1）创建一个空的游戏对象（单击GameObject > Create Empty命令）。把该游戏对象命名为BlueGoal，并给它分配Blue 标签。然后把游戏对象定位于(1.6, 2, 1.6)处。

（2）把一个盒子碰撞器附加到球门上，并选中Is Trigger 属性。把盒子碰撞器的大小改为(1.5, 1.5, 1.5)。

（3）把一个灯光附加到球门上（单击Component > Rendering > Light 命令）。把该灯光设置为点光源，并使之具有与球门对应的颜色，如图11.9所示。然后把灯光的强度改为3。

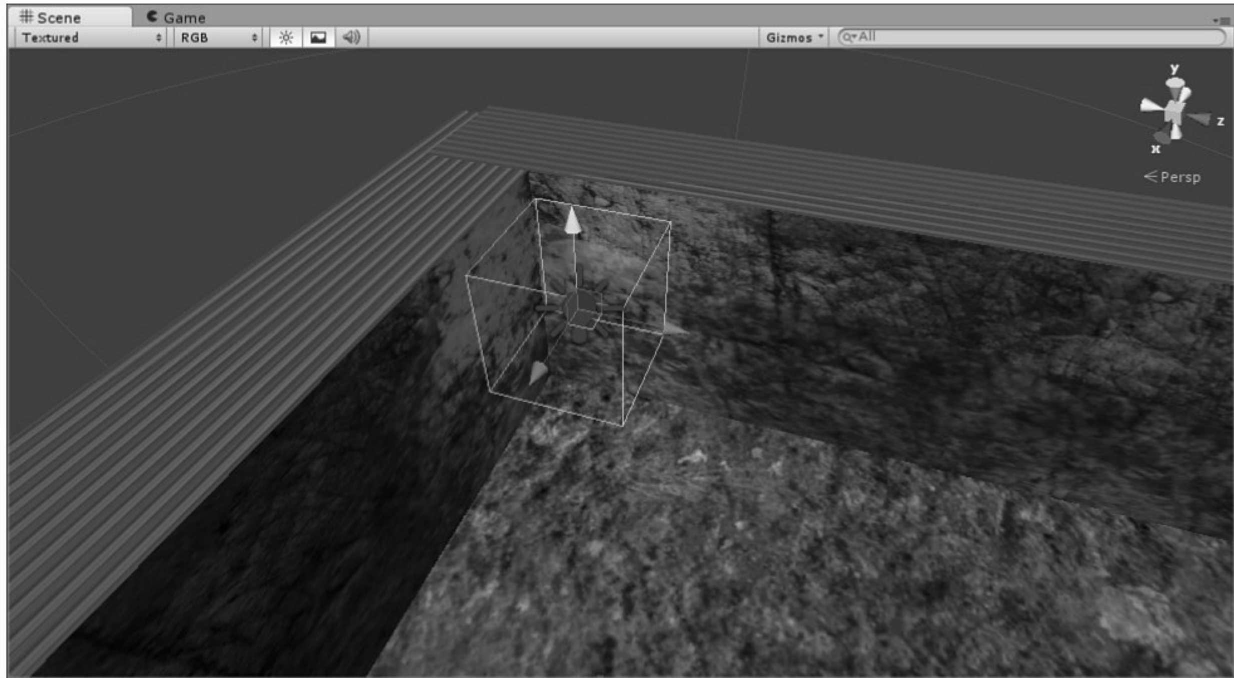


图11.9 蓝色球门

接下来，需要创建一个名为GoalScript的脚本，并把它附加到蓝色球门上。程序清单11.2显示了脚本的内容。

程序清单11.2 GoalScript.cs

```
using UnityEngine;
using System.Collections;
public class GoalScript : MonoBehaviour {
    private bool solved = false;
    // Use this for initialization
    void Start () {
    }
    // Update is called once per frame
    void Update () {
    }
    void OnTriggerEnter(Collider other)
    {
```

```
        if(other.tag == tag)
        {
            solved = true;
            other.rigidbody.isKinematic = true;
        }
    }
    public bool IsSolved()
    {
        return solved;
    }
}
```

在脚本中可以看到，OnTriggerEnter( )方法将相对于它自己的标签来检查与之接触的每个对象的标签。如果它们匹配，就把对象设置为不活动状态，并把那个球门标记为solved。

注意：

私有变量

你可能注意到，GoalScript 具有一个私有变量 solved 和一个公共方法IsSolved( )，该方法简单地返回这个变量。你可能感到疑惑的是，当可以简单地把这个变量设置为公共变量时，为什么还要做额外的工作。其原因是这样可以阻止其他任何对象或脚本意外地把球门设置为solved。由于只有球门可以访问那个变量，因此没有什么能够把它弄乱。这个方法只用于告诉游戏控制何时球门已完成。

当脚本完成并且附加到球门上时，就应该复制它。要创建其他的球门，可以遵循下面这些步骤。

（1）复制 BlueGoal。给新球门提供一个与其颜色对应的名称：RedGoal、GreenGoal 和OrangeGoal。

（2）把球门的标签改为与其对应的颜色。

（3）把点光源的颜色改为与球门对应的颜色。

（4）定位球门。可以设置任何一个角的颜色，只要每个球门都具有它自己的角即可。另外3 个角的位置是(1.6, 2, 48.4)、(48.4, 2, 1.6)和(48.4, 2, 48.4)。

现在应该设置了所有的球门，并且它们都可以工作。

11.4.2 游戏控制器

完成游戏所需要的最后一个元素是游戏控制器。这个控制器将负责在每一帧检查每个球门，并且确定全部4个球门何时都被标记为solved。对于这款特定的游戏，游戏控制器非常简单。要创建游戏控制器，可以遵循下面这些步骤。

（1）向场景中添加一个空的游戏对象，把它移到某个不碍事的位置，并把它重命名为GameController。

（2）创建一个名为GameControlScript的脚本，并把程序清单11.3中的代码添加到其中。然后把该脚本附加到游戏控制器上。

（3）选取游戏控制器，单击并把每个球门拖到Game Control Script组件上与它们对应的属性上，如图11.10所示。

程序清单11.3 游戏控制脚本

```
using UnityEngine;
using System.Collections;
public class GameControlScript : MonoBehaviour {
   public GoalScript red;
   public GoalScript blue;
   public GoalScript orange;
   public GoalScript green;
   private bool isGameOver = false;
```

```
    // Use this for initialization
    void Start () {
    }
    // Update is called once per frame
    void Update () {
        if(red.IsSolved() && blue.IsSolved() && orange.IsSolved() &&
green.
        IsSolved())
        {
            isGameOver = true;
        }
    }
    void OnGUI()
    {
        if(isGameOver)
        {
            GUI.Box(new Rect(Screen.width / 2 - 100,
                Screen.height / 2 - 50, 200, 75), "Game Over");
            GUI.Label(new Rect(Screen.width / 2 - 30,
                Screen.height / 2 - 25, 60, 50), "Good Job!");
        }
    }
}
```

在前面的脚本中可以看到，游戏控制器具有指向每个球门的引用。在每一帧上，控制器都会检查球门，看看它们是否已经完成。如果是，控制器就会把变量isGameOver设置为true，并且在屏幕上显示游戏结束的消息。

祝贺你，现在完成了Chaos Ball 游戏！

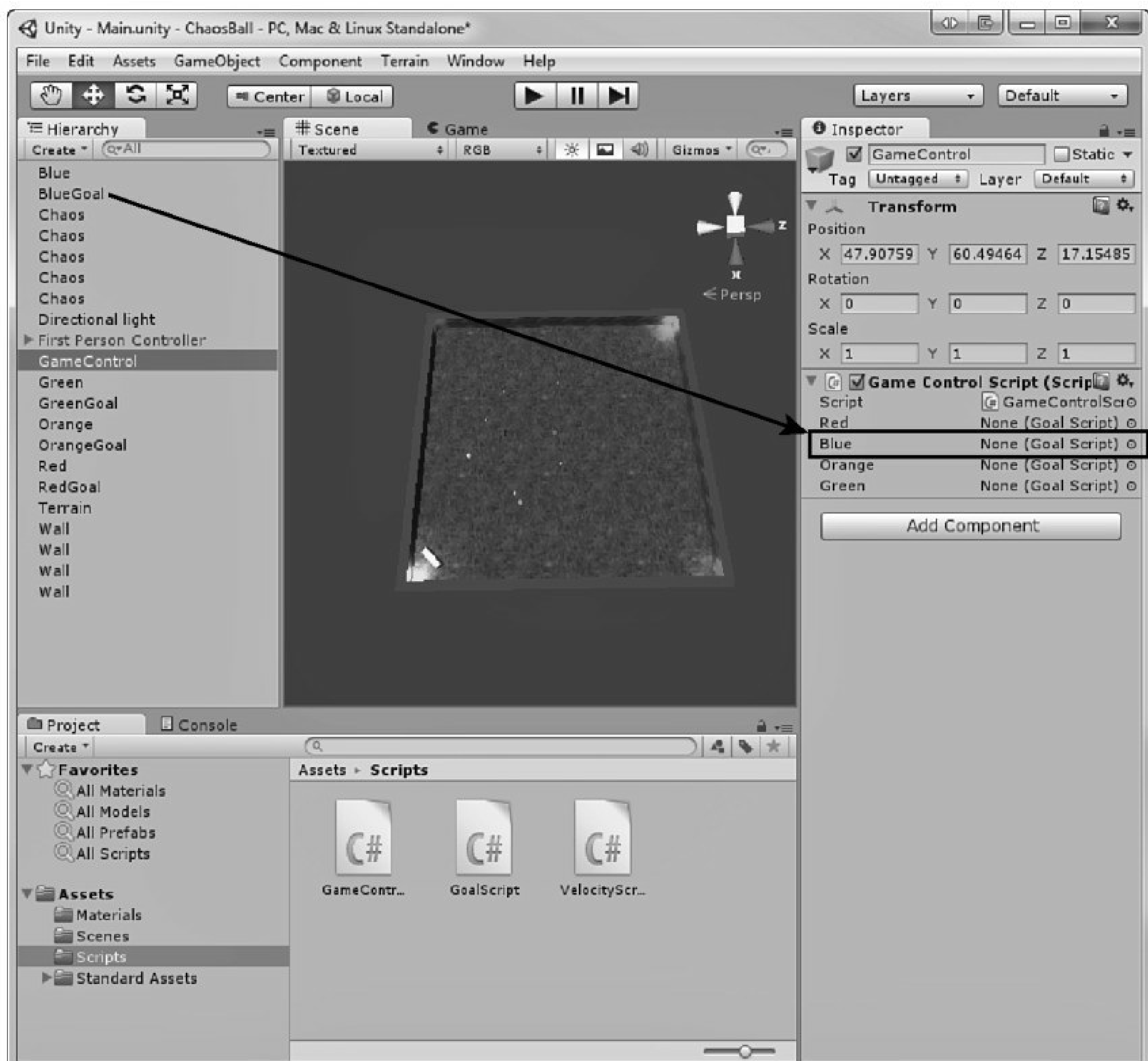


图11.10 把球门添加到游戏控制器上

11.5 改进游戏

即使Chaos Ball 是一个完整的游戏，可它还不够好。这里省略了几个特性，它们可以极大地改进游戏玩法。之所以省略了它们，是为了使你可以试验游戏，并使之变得更好。在某种程度上，可以说Chaos Ball现在是一个完整的原型。它是一个可以玩的游戏示例，但它还要进行完善。我们鼓励你再从头开始阅读一遍本章的内容，并且寻找一些可以使游戏变得更好的方式。在你玩游戏时，可以自己思考一下以下问题。

游戏太容易还是太难？

什么将使它变得更容易或者更难？

什么可以给游戏提供令人兴奋的因素？

游戏的哪些部分很有趣，哪些部分又比较乏味？

在下面的练习中，你将有机会改进游戏，并添加其中一些特性。注意，如果你遇到任何错误，就意味着你遗漏了一个步骤。一定要回过头来把所有的一切都复查一遍，以解决可能发生的任何错误。

11.6 小结

在本章中，你制作了ChaosBall游戏。你首先设计了游戏，确定了理念、规则和需求。接着，你雕刻了舞台，并且知道了有时可以制作单个对象并复制它以节省时间。之后，你创建了玩家、混乱球、彩球、球门和游戏控制器。在本章最后，尝试了玩游戏，并且考虑了改进它的方式。

11.7 问与答

问：我们为什么对混乱球使用连续的碰撞检测？我认为这会降低性能。

答：事实上，连续的碰撞检测可能会降低性能。不过，在这种情况下，需要这样做。混乱球比较小且速度快，足以使它们有时能够穿过墙壁。

问：球门基于球的标签来确定是否有正确的球进入了球门。只利用球的名称可以做到同样的事情吗？

答：绝对可以！使用标签的原因在于它可以使事情变得简单。使用标签和编辑器，可以编写通用的脚本。这允许创建脚本一次，然后把它使用4次。

11.8 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

11.8.1 问题

1．判断题：这款游戏使用了Unity的弹性物理材质。

2．玩家怎样会输掉游戏？

3．所有的球对象都会冻结在哪根位置轴上？

4．判断题：球门利用OnTriggerEnter( )方法确定一个对象是否是正确的球。

5．为什么省略了一些基本的特性？

11.8.2 答案

1．错误。你创建了自己的超级弹性物理材质。

2．这是一个有意捉弄人的问题。玩家不能输掉游戏。

3．y轴。

4．正确。

5．给读者提供一个添加它们的机会。

11.8.3 练习

关于制作游戏的最佳部分是：你可以像自己所想的那样制作它们。遵循指导可能是一种良好的学习经历，但是将不能获得制作自定义游戏的满足感。在这个练习中，你将有机会稍稍修改游戏，使某些方面更独特。至于你想怎样修改游戏则完全取决于你自己。下面给出了一些建

议。

尝试添加一个按钮，无论何时游戏完成，它都允许玩家再玩一遍（我们还没有介绍GUI元素，但是在最后一款游戏中存在这个特性，看看你是否可以弄明白它）。

尝试添加一个计时器，使得玩家知道赢得游戏花了多长的时间。

尝试添加混乱球的变体。

尝试添加一个混乱球门，它需要所有的混乱球才能完成。

尝试更改玩家的减震器的大小或形状。尝试制作一个源于许多形状的复杂的减震器。

第12章 预设

在本章中你将学到：

预设的基础知识；

怎样处理自定义的预设；

怎样在代码中实例化预设。

预设是一个复杂的对象，它被打包起来，以便用户可以轻松地反复重建它。在本章中，你将学习关于预设的所有知识。首先将了解预设以及它们可以做什么。接着，将学习如何在Unity中创建预设。你将学习继承的概念。最后将学习怎样通过编辑器和代码向场景中添加预设。

12.1 预设的基础知识

如前所述，预设是打包游戏对象的一类特殊的资源。与在Hierarchy视图中简单地嵌套对象不同，预设存在于Project视图中，并且可以跨许多场景反复重用。这使你能够构建复杂的对象，比如敌人，以及使用它构建一支军队。也可以利用代码创建预设，这允许在运行时生成几乎无限数量的对象。最大的优点是可以把任何游戏对象或者游戏对象的集合放入预设中，让一切都变成可能！

注意：

思考练习

如果你在理解预设的重要性方面遇到麻烦，可以考虑下面的情况：在上一章中，你制作了Chaos　Ball游戏。在制作该游戏时，不得不制作单个混乱球，并把它复制4次。如果你希望在运行时能够自由地制作更多的混乱球，则该如何？事实是原来的办法不再适用了。如果没有预设，则无论如何也做不到这一点。现在，如果具有一款使用魔兽类型的敌人的游戏时，则该如何？同样，可以建立单独一只魔兽，然后把它复制许多次，但是如果希望在另一个场景中再次使用魔兽，则该如何？你将不得不在新场景中完全重新制作魔兽。不过，如果魔兽是一个预设，那么它将是项目的一部分，并且可以在任意数量的场景中再次重用它。预设是Unity游戏开发的一个重要方面。

12.1.1 预设的术语

在与预设打交道时，知道一些术语很重要。如果你熟悉面向对象程序设计实践的概念，可能会注意到一些相似之处。

预设（prefab）：预设是基本对象，它只存在于Project 视图中，可以把它视作蓝图。

实例（instance）：预设在场景中的实际对象。如果预设是汽车的蓝图，那么实例就是实际的汽车。如果把Scene视图中的对象称为预设，这实际上意味着它是一个预设实例。预设的实例（instance ofa prefab）这个短语是预设的对象（object ofaprefab）的同义词。

实例化（instantiate）：创建预设实例的过程。它是一个动词，用法如下："我需要实例化这个预设的实例。"

继承（inheritance）：它与标准的程序设计中的继承不是一回事。在这里，术语"继承"指的是预设的所有实例用于链接到预设本身的性质。在本章后面将更详细地介绍它。

12.1.2 预设的结构

无论你是否知道，你都已经在与预设打交道。Unity 的字符控制器就是一个预设。要把预设的对象实例化到某个场景中，只需单击并把它拖到Scene视图或Hierarchy视图中的某个位置即可，如图12.1所示。

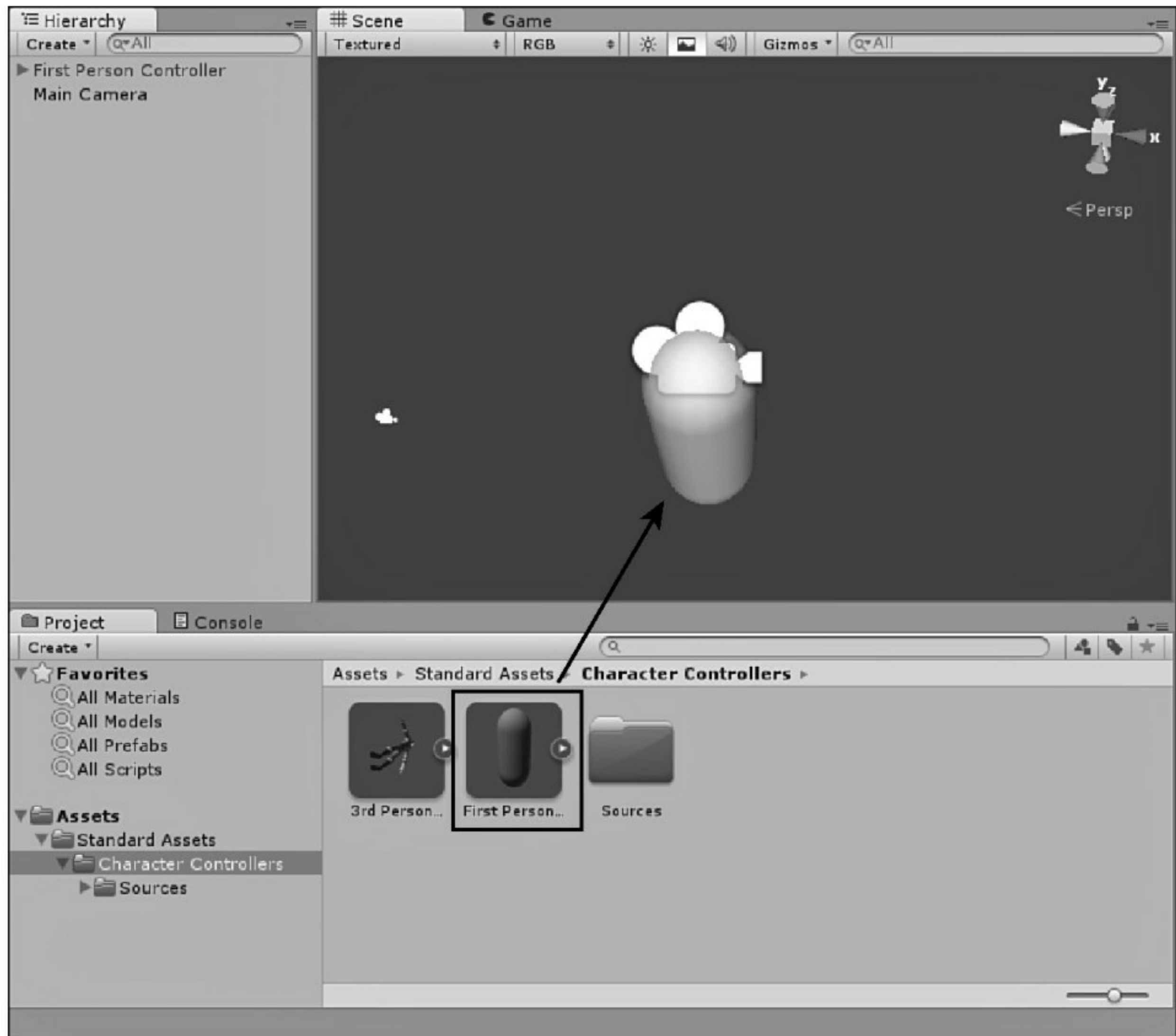


图12.1 把预设实例添加到场景中

在查看 Hierarchy 视图时，总是可以辨别哪些对象是预设的实例，因为它们将以蓝色显示，如图12.2所示。就像非预设的复杂对象一样，预设的复杂实例也具有一个箭头，允许展开它们以及修改其内的对象。



图12.2 在 Hierarchy 视图中以蓝色显示的预设实例

由于预设是一种属于项目而不是属于特定场景的资源，因此将在Project 视图中编辑预设。就像游戏对象一样，预设可能很复杂。通过单击预设右边的箭头，来编辑预设的子元素，如图12.3所示。单击这个箭头将展开对象以便进行编辑，再次单击该箭头将把预设再次收缩起来。



图12.3 在 Project 视图中展开预设的内容

12.2 处理预设

使用 Unity 的内置预设很不错，但是通常你将希望创建自己的预设。创建预设是一个包含两个步骤的过程：第一步是创建预设资源；第二步是利用一些内容填充资源。

创建预设实际上很容易。就像所有其他的资源一样，你将希望首先在 Project 视图中的Assets 下面创建一个文件夹来包含它们。然后，只需右键单击新创建的文件夹并选择Create >Prefab命令即可，如图12.4所示。新的预设将显示出来，可以给它提供想要的任何名称。由于预设是空的，它将显示为一个空的白框。

下一步是利用一些内容填充预设。可以把任何游戏对象都放入预设中。只需在Scene视图中创建对象一次，然后单击并把它拖到预设资源上。

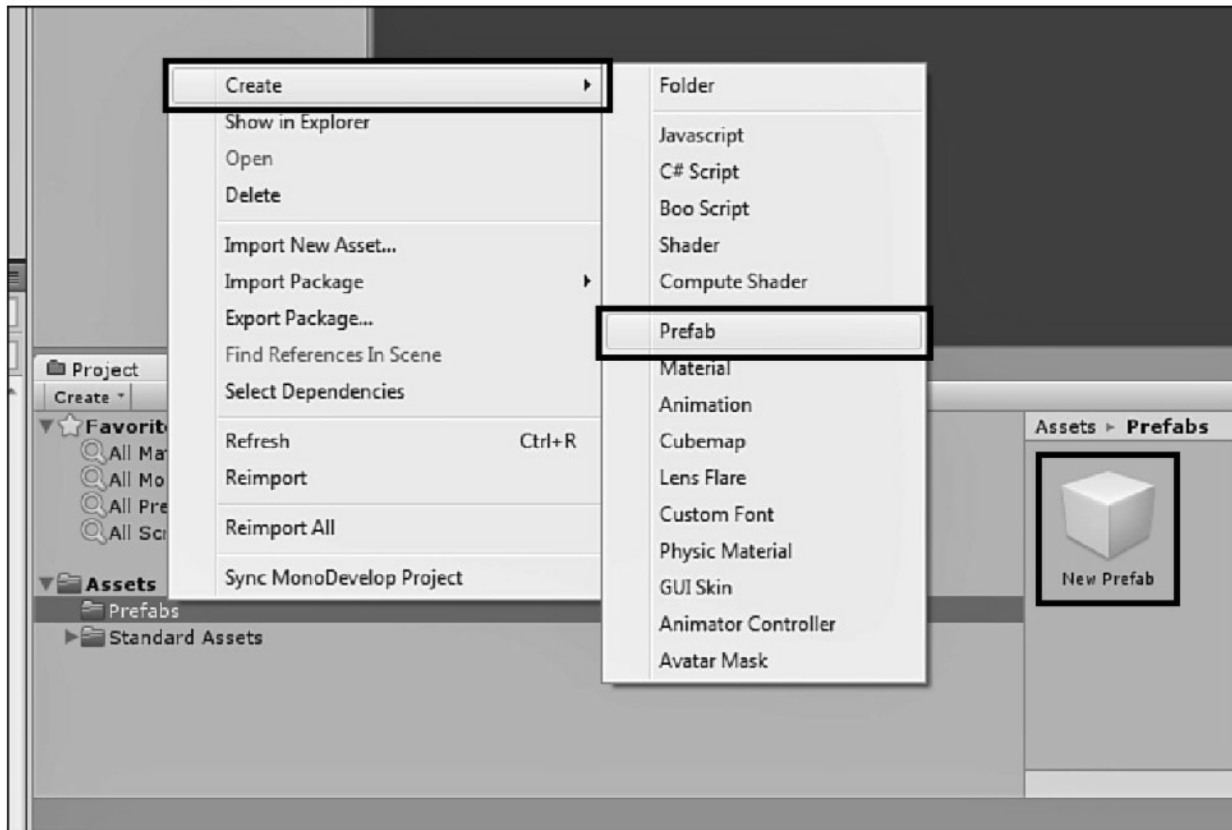


图12.4 创建新的预设

创建预设

让我们创建一种预设资源，并利用复杂的游戏对象填充它。这里创建的预设资源将在本章后面使用，因此不要删除它。

（1）创建一个新的项目或场景，并向场景中添加一个立方体和一个球体。

（2）把立方体定位于(0, 0, 0)处，并把它缩放为(.5, 2, .5)。然后给立方体添加一个刚体。把球体定位于(0, 1.2, 0)处，并把它缩放为(.5, .5, .5)。然后把一个点光源组件放在球体上。

（3）在 Hierarchy 视图中单击并把球体拖到立方体上，这将把球体嵌套在立方体中，如图12.5所示。

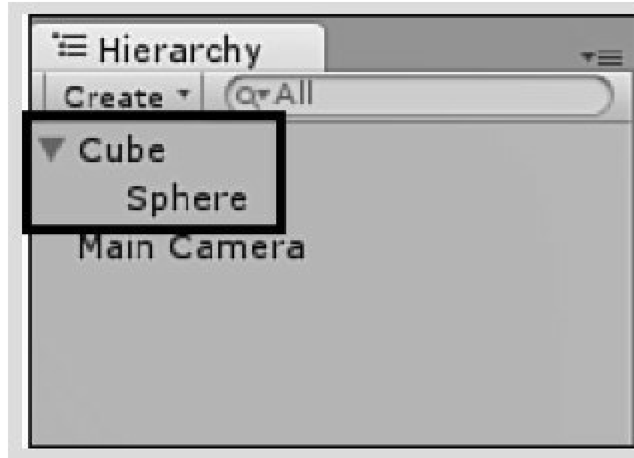


图12.5 嵌套在立方体下的球体

（4）在 Project 视图中的 Assets 文件夹下创建一个新文件夹，并把新文件夹命名为Prefabs。在Prefabs 文件夹中创建一个新的预设（右键单击并选择Create > Prefab 命令），并把新预设命名为Lamp。

（5）在 Hierarchy 视图中，单击并把立方体（包含球体）拖到Project 视图中的 Lamp预设上，如图 12.6 所示。你将注意到该预设现在看起来像一盏灯，还将注意到 Hierarchy视图中的立方体和球体变为蓝色。此时，可以从场景中删除立方体和球体，它们现在包含在预设中。

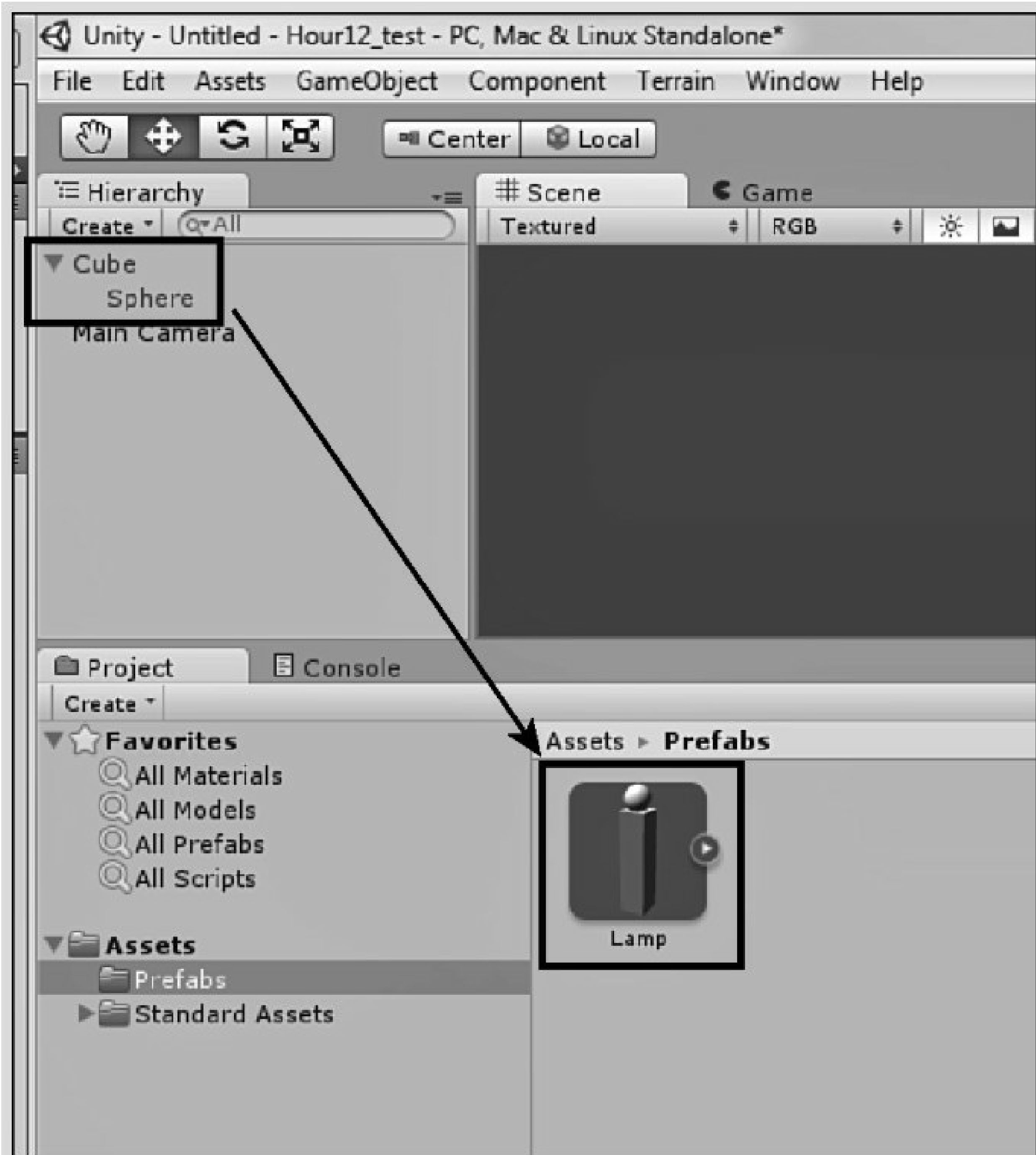


图12.6 给预设添加对象

12.2.1 向场景中添加预设实例

一旦创建了预设资源，就可以根据需要把它向某个场景中添加许多

次，或者添加到项目中的任意数量的场景中。要把预设实例添加到场景中，只需从Project视图中单击预设并把它拖到Scene视图中的合适位置即可。你将注意到，在把预设实例放入场景中时，可以轻松地把实例放在其他对象顶部。这使得定位新实例非常简单。

创建多个预设实例

在上一个练习中，你创建了一个Lamp预设。这一次，你将使用该预设在场景中创建许多盏灯。一定要保存这里创建的场景，在本章后面将使用它。

（1）在用于上一个练习的相同项目中创建一个新的场景。

（2）向场景中添加一个立方体，然后把该立方体定位于(0,　0,　0)处，并把它缩放为(5, .1, 5)。

（3）从Prefabs文件夹中单击并把Lamp预设拖到平坦的立方体上，如图12.7所示。根据需要把这个动作重复许多次。注意可以轻松地把灯放在立方体上，并且定位也相当简单。还要注意对象名称不再是Cube，它现在是Lamp，就像预设一样。

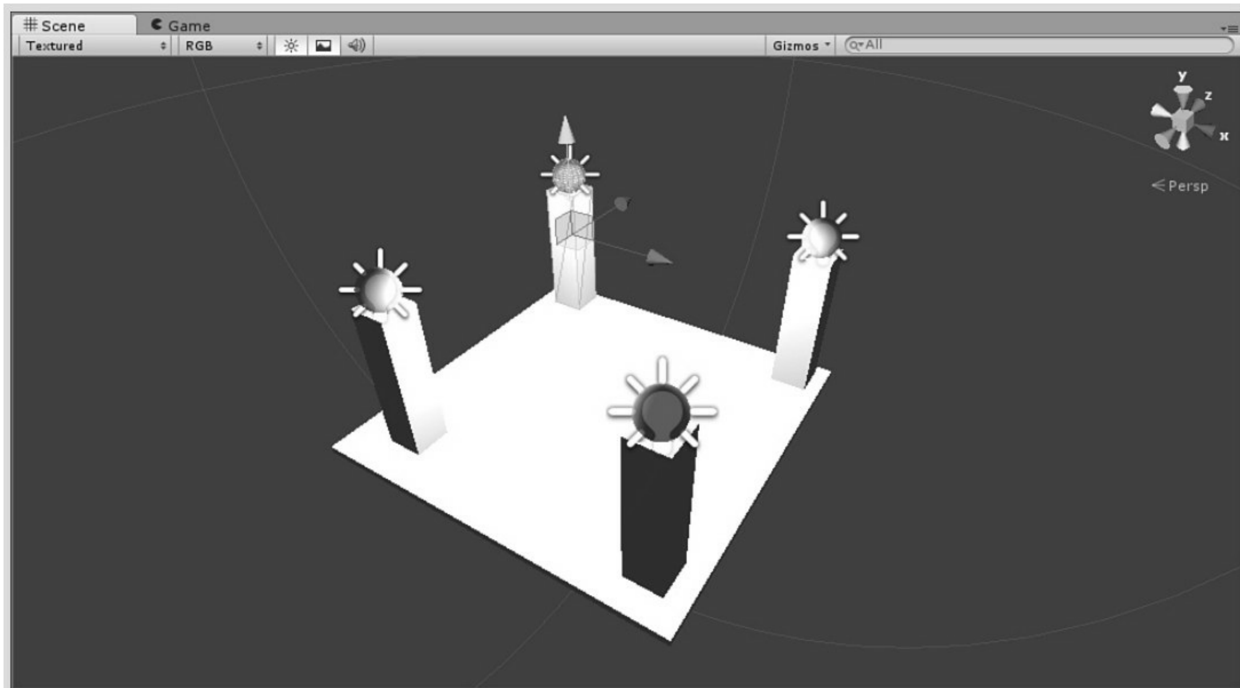


图12.7 把灯放在场景中

12.2.2 继承

当把术语继承（inheritance）与预设一起使用时，它意指一种链接，并且预设的实例将通过它连接到实际的预设资源。也就是说，如果更改预设资源，也会自动更改预设的所有对象。这是极其有用的。你往往会把大量的预设对象放入场景中，只是因为意识到它们都需要一点微小的改变。如果没有继承，将不得不独立更改每个对象。

可以用两种方式更改预设资源。第一种是在Project视图中执行更改。只需在Project视图中选取预设资源，即可在Inspector视图中调出它的组件和属性。如果需要修改子元素，可以展开预设（如前所述），并以类似的方式更改那些对象。

可以修改预设资源的另一种方式是把一个实例拖到场景中。在这里，可以执行任何想要的重大修改。完成后，只需把该实例拖回预设资源上以更新它即可。

更新预设

迄今为止，你创建了一个预设，并向场景中添加了几个实例。现在，你将有机会修改预设，并且查看它如何影响场景中已经存在的资源。这个练习将使用在上一个练习中创建的场景。如果你还没有完成那个练习，将需要完成该练习，然后才能继续下面的操作。

（1）打开你在前面创建的带有灯的场景。

（2）从Project视图中选择Lamp预设并展开它（单击右边的箭头），并且选择Sphere子组件。在Inspector视图中，把灯光的颜色改为橙色，如图12.8所示。注意场景中的预设是怎样自动改变的。

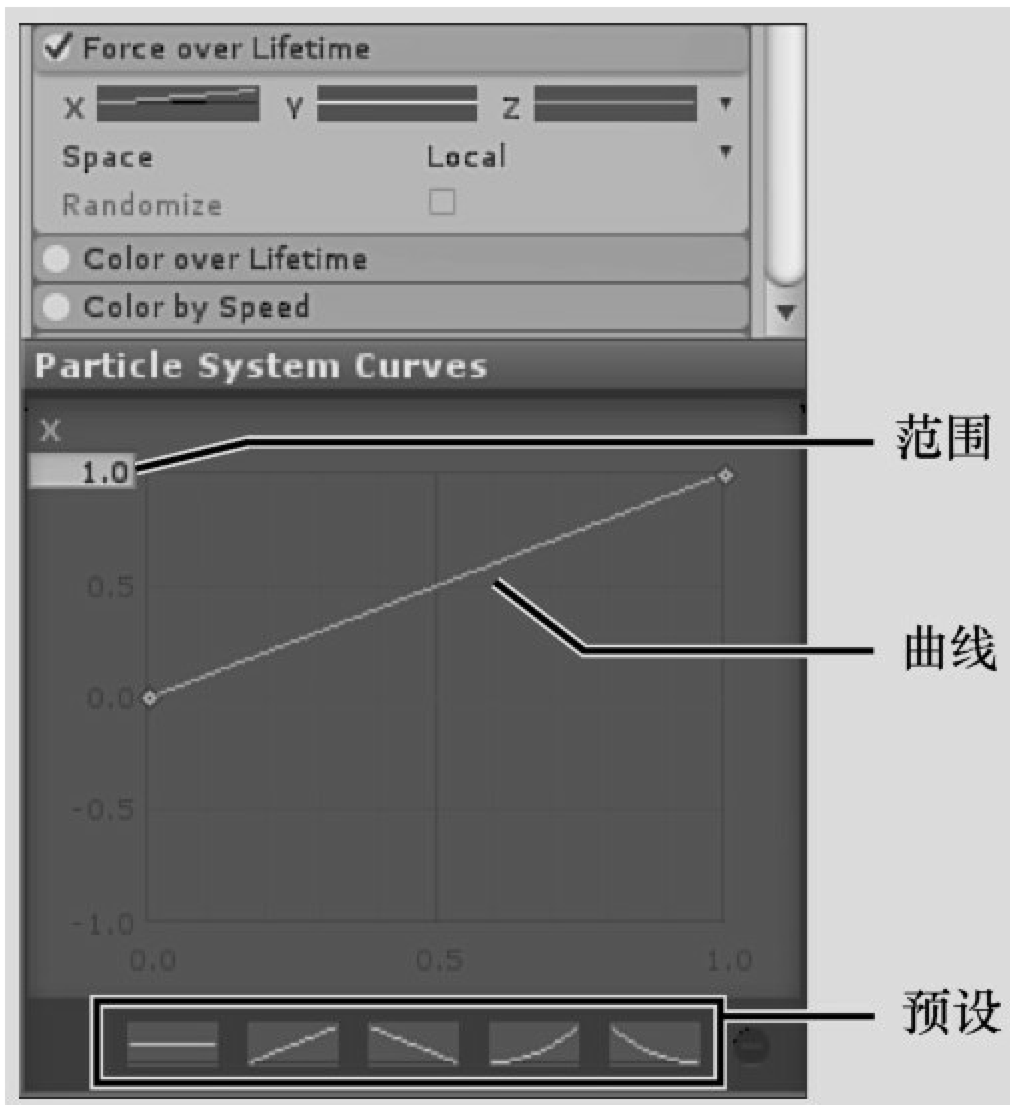


图12.8 修改过的灯实例

（3）选取场景中的灯实例之一。在 Hierarchy 视图中单击其名称左边的箭头以展开它，并且选取Sphere子对象。然后把球体的灯光改回白色。注意其他预设对象没有改变。

（4）单击修改过的灯实例，并把它拖回到预设资源上，如图12.9所示。注意所有的实例如何变回白色灯光。

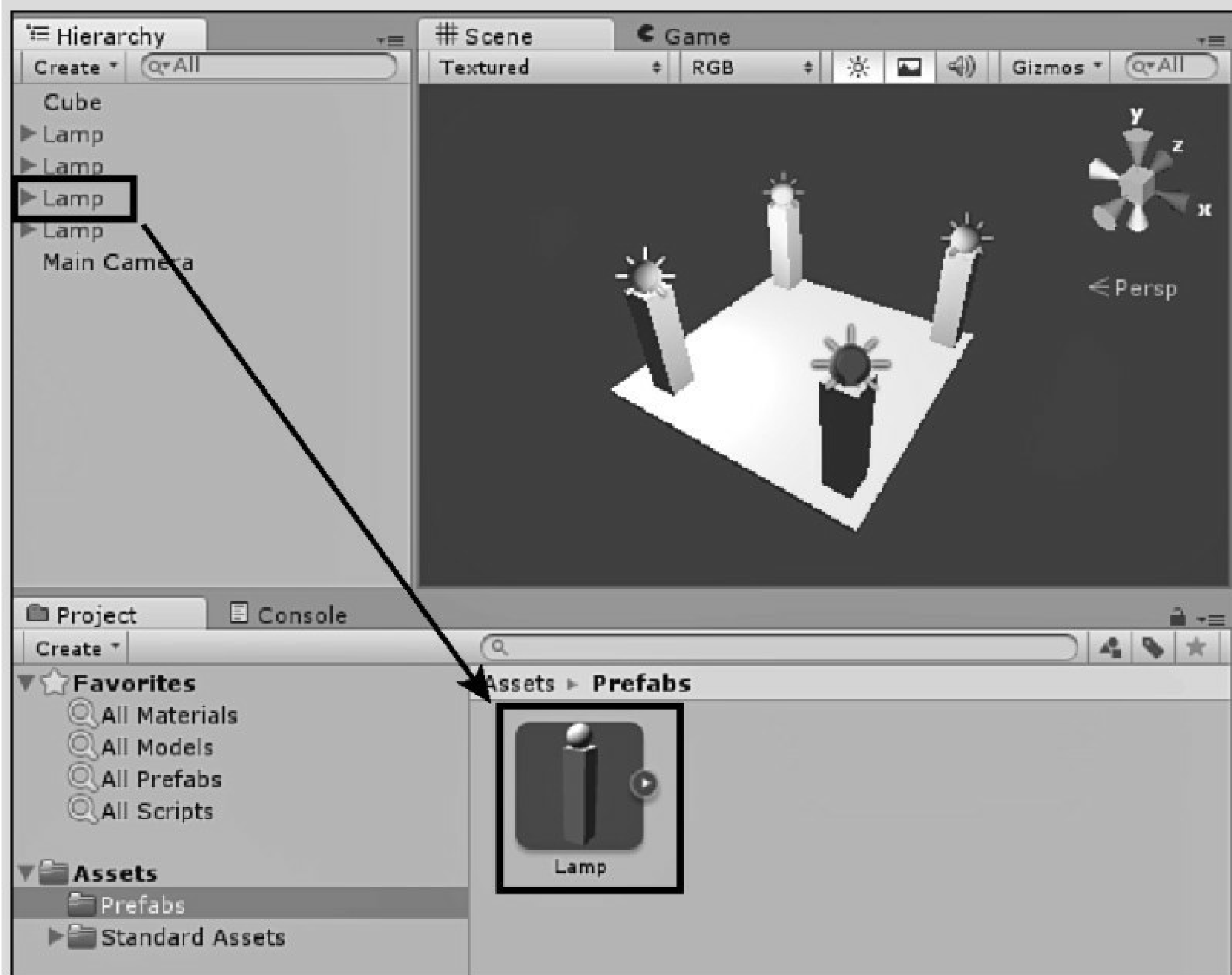


图12.9 利用修改过的实例更新Lamp预设

12.2.3 中断预设

有时，需要中断预设实例与预设资源之间的链接。如果需要一个预设的对象，但是在预设改变时不希望对象也改变，就可能需要中断它们之间的链接。中断实例与预设之间的链接不会以任何方式改变实例，它仍然会保持它的所有对象、组件和属性。唯一的区别是：它将不再是预设的实例，因此不再会受到继承的影响。

要中断对象与预设资源之间的链接，可以在 Hierarchy 视图中简单地选取对象。在选取它之后，单击GameObject > Break Prefab Instance命令。你将注意到对象没有改变，但是它的名称从蓝色变为黑色。一旦链

接中断，将不能重新应用它。

12.3 通过代码实例化预设

把预设对象放入场景中是构建一致的、有计划的关卡的极佳方式。不过，有时，你希望在运行时创建实例。你可能希望敌人复活，或者希望随机地摆放它们。也有可能是你需要如此多的实例，以至于不再能够手工摆放它们。无论出于什么原因，通过代码实例化预设都是一种良好的解决方案。

有两种方式实例化场景中的预设对象，并且它们都使用Instantiate()方法。使用Instantiate( )的第一种方式如下：

```
Instantiate(GameObject prefab);
```

可以看到，该方法简单地读入一个游戏对象变量，并创建它的一个新对象。新对象的位置、旋转和缩放与Project 中的预设相同。使用Instantiate( )方法的第二种方式如下：

```
Instantiate(GameObject prefab, Vector3 position, Quaternion rotation);
```

这个方法需要3个参数。第一个参数仍然是要制作副本的对象，第二个和第三个参数是想要的新对象的位置和旋转角度。你可能注意到旋转角度存储在一个名为Quaternion的变量中，只需知道这是Unity存储旋转信息的方式即可。Quaternion的真正应用超出了本章的范围。在本章末尾的练习中可以找到在代码中实例化对象的两种方法的示例。

12.4 小结

在本章中，你学习了Unity中的预设的相关知识。你首先学习了预设的基础知识：概念、术语和结构。接着，你学习了创建自己的预设，还探索了如何创建它们，把它们添加到场景中，修改它们以及中断它们。最后，你了解了通过代码实例化预设对象。

12.5 问与答

问：预设似乎很像面向对象程序设计（object oriented programming，OOP）中的类，这种说法准确吗？

答：是的，类与预设之间有许多相似之处。它们都像蓝图，它们的对象都是通过实例化创建的，并且都链接到原型。

问：一个场景中可以具有多少个预设的对象？

答：可以根据需要具有许多个这样的对象。不过，要知道的是，在超过某个数字后，游戏的性能将会受到影响。每次创建一个实例时，它都将持久存在，直到销毁它为止。因此，如果创建10000个对象，场景中将有10000个对象存在。

12.6 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

12.6.1 问题

1．哪个术语是用于创建预设资源的实例的？

2．用于修改预设资源的两种方式是什么？

3．什么是继承？

4．可以用多少种方式使用Instantiate( )方法？

12.6.2 答案

1．实例化。

2．可以通过Project视图修改预设资源，也可以在Scene视图中修改实例，并把它拖回到Project视图中的预设资源上来修改它。

3．继承是把预设资源连接到其实例的链接。它实质上意味着当资源改变时，对象也会改变。

4．两种。可以只指定预设，或者也可以指定位置和旋转角度。

12.6.3 练习

在这个练习中，你将再次处理在本章前面创建的预设。这一次，将通过代码实例化预设的对象，并且希望利用它产生一些乐趣。在用于第12章（Hour 12）的本书配套资源中可以找到用于这个练习的完整项目，其名称为Hour12_Exercise。

1．在Lamp预设所在的相同项目中创建一个新的场景。在Project视

图中单击Lamp预设，并把它置于(−1, 1, −5)处。

2．向场景中添加一个空的游戏对象，把它重命名为SpawnPoint，并放置在(1，1，−5)处。向场景中添加一个平面，然后把它放置在(0，0, −4)处，并把旋转角度设置为(270, 0, 0)。

3．向项目中添加一个脚本，把该脚本命名为PrefabGenerator，并把它附加到复活点对象上。程序清单12.1具有预设生成器脚本的完整代码。

程序清单12.1 PrefabGenerator.cs

```
using UnityEngine;
using System.Collections;
public class PrefabGenerator : MonoBehaviour {
  //We will store a reference to the target prefab from the inspector
  public GameObject prefab;
  // Use this for initialization
  void Start () {
  }
  // Update is called once per frame
  void Update () {
    //Whenever we hit the B key, we will generate a prefab at the
    //position of the original prefab
    //Whenever we hit the space key, we will generate a prefab at the
    //position of the spawn object that this script is attached to
    if(Input.GetKeyDown(KeyCode.B))
      Instantiate(prefab);
    if(Input.GetKeyDown(KeyCode.Space))
      Instantiate(prefab, transform.position, transform.rotation);
  }
```

}

4．选取复活点，把Lamp预设拖到Prefab Generator 组件的Prefab 属性上。现在运行场景。注意：按下B键将在默认的预设位置创建一盏灯，而按下空格键则会在复活点创建一个对象。还要注意预设之间相互碰撞将如何引发一些独特的交互。

你可能注意到，在运行这个场景时，创建的灯将持续不断地一直往下坠落，并且永远不会从场景中消失。作为一个额外的挑战，看看你是否可以在场景下面创建一个带有触发器的平面，它可以在灯进入时销毁它们。这样，游戏就可以在灯不再可见时清理它们，从而避免了长期运行后可能出现的性能问题。

第13章 图形用户界面

在本章中你将学到：

Unity GUI 的基础知识；

怎样使用不同的GUI 控件；

怎样自定义GUI。

图形用户界面（graphical user interface，GUI）是一组特殊的组件，它们负责给用户发送信息以及读取来自用户的信息。在本章中，你将学习有关使用Unity的内置GUI系统的知识。你首先将学习 GUI 的基础知识。接着，将开始试验多个 GUI 控件。最后，将学习如何使用样式和皮肤自定义GUI的外观。

13.1 GUI的基础知识

如前所述，图形用户界面（通常称为GUI）是位于玩家和实际游戏之间的一个特殊的层。GUI的作用是给用户显示重要的信息，有时也会从用户那里读回数据。在Unity中，GUI由利用代码创建的多个控件组成，这些控件是诸如标签、按钮、文本框和滑块之类的元素。后面将更详细地介绍控件。

提示：

GUI设计

一般说来，你会希望提前设计GUI。需要更多地考虑在屏幕上将显示什么数据，在什么地方显示它们，以及如何显示它们。信息太多将让人感觉屏幕显得混乱不堪，信息太少又会使玩家感到糊涂或者不确信。总是要寻找一些方式来浓缩信息或者使信息更有意义，这样你的玩家将会感谢你。

警告：

创建GUI

由于GUI是通过代码创建的，可以把它添加给任何脚本和任何对象。如果你不小心，这可能会导致内容组织方面的问题。把一点GUI代码放在多个脚本中可能使查找你想要的部分变得困难。此外，错误也会变得更难跟踪和修正。把GUI部分放在多个对象上也会使得难以定义哪个对象负责GUI。处理这种情况的一个好办法是把所有的GUI代码都放在一起，它就是一个专门指定的对象。把所有的GUI材料都放在同一个位置，将使游戏开发变得更容易。

要把GUI添加到项目中，将需要把一个特殊的函数添加到场景中的脚本中。至于把它放在哪个脚本中是无关紧要的。只要脚本是活动的并

且位于运行的场景中的某个游戏对象上，你就会看到GUI。创建GUI 的方法是使用OnGUI( )，它看起来如下：

```
void OnGUI()
{
   //GUI code goes here
}
```

可以看到，OnGUI( )方法不带参数，并且不返回数据。就像Update( )一样，在每一帧上都会调用OnGUI( )方法，并把GUI 组件绘制到屏幕上。如你将在本章后面看到的，GUI 控件由简单的代码行组成，用于那些控件的代码位于OnGUI( )方法内。

创建GUI

让我们把一个基本的GUI控件绘制到屏幕上。在这个练习中，你将把带有一个标签的消息写到屏幕上。不要过于关注标签代码，在后面将更详细地介绍它。作为替代，只需确保你可以使GUI出现在屏幕上，以便为下一节做好准备。

（1）创建一个新的项目或场景。然后创建一个名为BasicGUIScript的新脚本，并把它附加到Main Camera 上。

（2）把以下代码添加到任何方法的外面，但是要出现在类里面（如果需要指导，可以参见用于第13章（Hour 13）的本书配套资源）：

```
void OnGUI()
{
   GUI.Label(new Rect(0, 0, 80, 20), "Hello World");
}
```

（3）运行场景。注意“Hello World”是如何打印到屏幕上的，如图13.1所示。

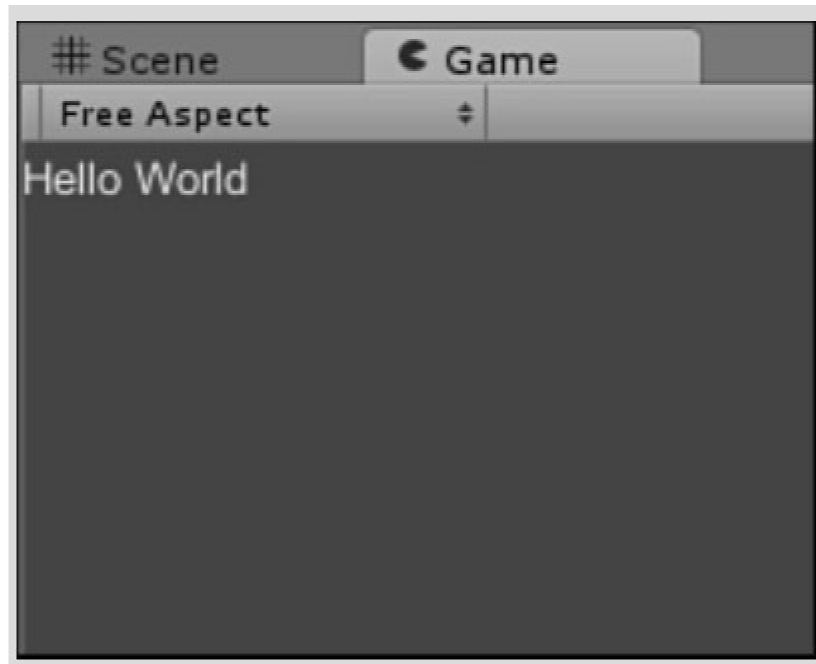


图13.1 把“Hello World”打印到屏幕上

13.2 GUI控件

在本节中，你将处理更常见的内置Unity GUI 控件。其中大多数控件是以类似的方式创建的，并且以类似的方式工作。不过，在深入研究特定的控件之前，需要熟悉Rect变量类型。Rect是rectangle（矩形）的简写，所有的组件都通过它来获知自己在屏幕上的位置。如本书前面所提到的，GUI元素只能使用2D坐标。因此，任何GUI元素的精确位置和大小都可以利用一个矩形来指定。在本章中，将多次看到以下代码：

```
new Rect(<left>, <top>, <width>, <height>)
```

上面的代码创建一个新的Rect，它包含左边的x轴位置的值、上边的y轴位置的值以及宽度值和高度值。因此，如果想要指定一个从左上角开始并且宽度和高度分别为100单位和50单位的矩形，可以编写以下代码：

```
new Rect(0, 0, 100, 50)
```

除了包含位置的Rect之外，每个控件还需要一些额外的信息，后面将单独介绍它们。

在处理每个组件之前需要知道的最后一件事是屏幕坐标是如何设置的。如前所述，屏幕只有两个尺寸。左上角是原点(0, 0)，右下角是最大屏幕尺寸。由于Unity可能同时处理许多不同的屏幕大小，很难准确知道最大屏幕大小是什么。因此，可以使用两个内置变量Screen.width和Screen.height，知道任何屏幕上的最大大小是什么。例如，用于创建一个Rect并使其左上角精确地位于屏幕中心的代码如下：

```
new Rect(Screen.width / 2, Screen.height / 2, 100, 50)
```

提示：

使控件出现在屏幕中心

通常，你希望控件准确地出现在屏幕中心。你可能注意到，在屏幕中间创建一个 Rect 实际上会把 Rect 置于屏幕中间偏下和偏右一点。这是由于Rect的左上角位于屏幕中间，其余部分则会正常延展。要把控件放在实际的中间位置，则需要更多一点数学知识。实质上，需要把 Rect 放在中间减去其一半宽度和高度的位置。这样，一半将位于中间的左上方，一半则位于右下方。因此，要把一个宽100、高50的Rect放在屏幕的中间，可以编写以下代码：

```
new Rect(Screen,width / 2–50, Screen.height / 2–25, 100, 50)
```

这最初看起来似乎有些令人糊涂，但是深入一点研究数字，它很快就会显得有意义了。

13.2.1 标签

标签控件是最基本的控件，它只用于把数据显示为字符串。创建标签的代码如下：

```
GUI.Label(new Rect(<x>, <y>, <w>, <h>), <Some String>);
```

因此，要在左上角创建一个标签，用于显示“Hello World”，可以编写以下代码：

```
GUI.Label(new Rect(0, 0, 80, 20), "Hello World");
```

在上一个练习中可以看到它的实际应用。

13.2.2 方框

方框控件类似于标签，唯一的区别是方框还具有包含标签的深色边框。方框可以用作多种其他控件的背景。用于创建方框的语法如下所示：

```
GUI.Box(new Rect(<x>, <y>, <w>, <h>), <Some String>);
```

因此，如果希望方框出现在屏幕中上方，并且显示“Box Label”，可

以编写以下代码：

```
GUI.Box(new Rect(Screen.width / 2 - 50, 0, 100, 50), "Box Label");
```

如果希望在相同位置显示一个不带标签的空方框，也可以编写如下代码：

```
GUI.Box(new Rect(Screen.width / 2 - 50, 0, 100, 50), "");
```

图13.2显示了利用前面的代码创建的方框。

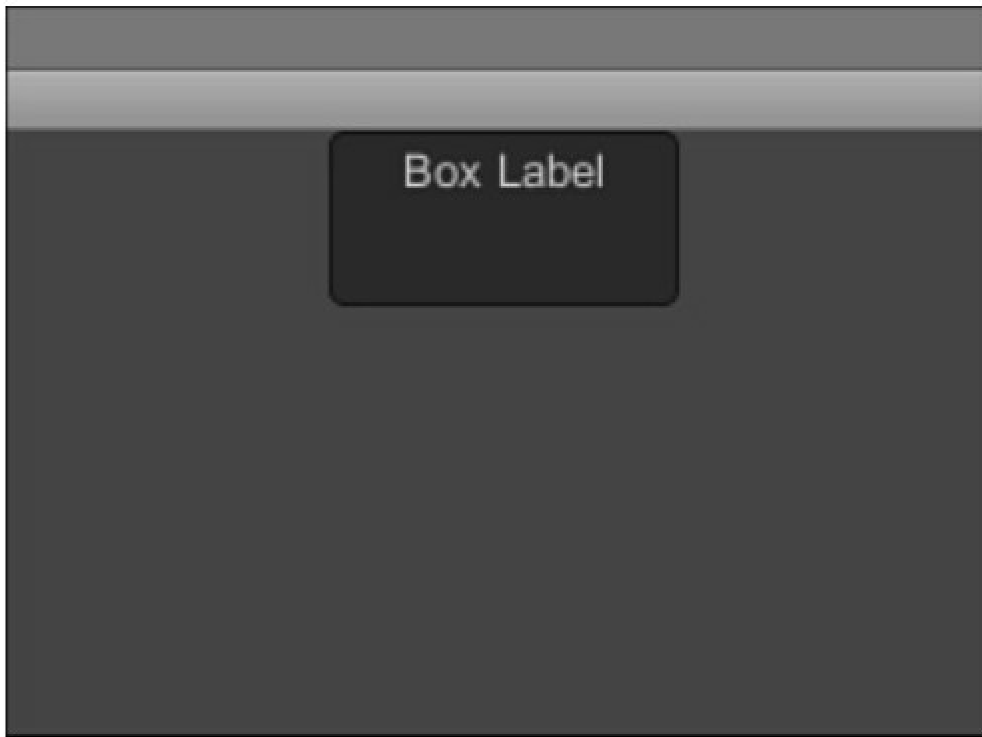


图13.2 方框控件

13.2.3 按钮

按钮是与条件语句协同工作的简单控件。按钮可以为“假”（未按下），或者为“真”（按下）。按钮控件每次只能按下一次，继续按住按钮将不会产生额外的作用。用于按钮的语法如下所示：

```
if(GUI.Button(new Rect(<x>, <y>, <w>, <h>), <Some String>))
{
```

```
    //whatever your button does when clicked.
}
```

因此，要把一个按钮放在屏幕的左上角，在单击它时把某个变量设置为“假”，可以输入以下代码：

```
if(GUI.Button(new Rect(0, 0, 40, 20), "Exit ?"))
{
    gameOver = true;
}
```

注意：

如果实际运行这段代码，它将会失败，因为 gameOver 不存在。对于这个示例，只会附带显示出按钮。图13.3显示了通过前面的代码创建的按钮。

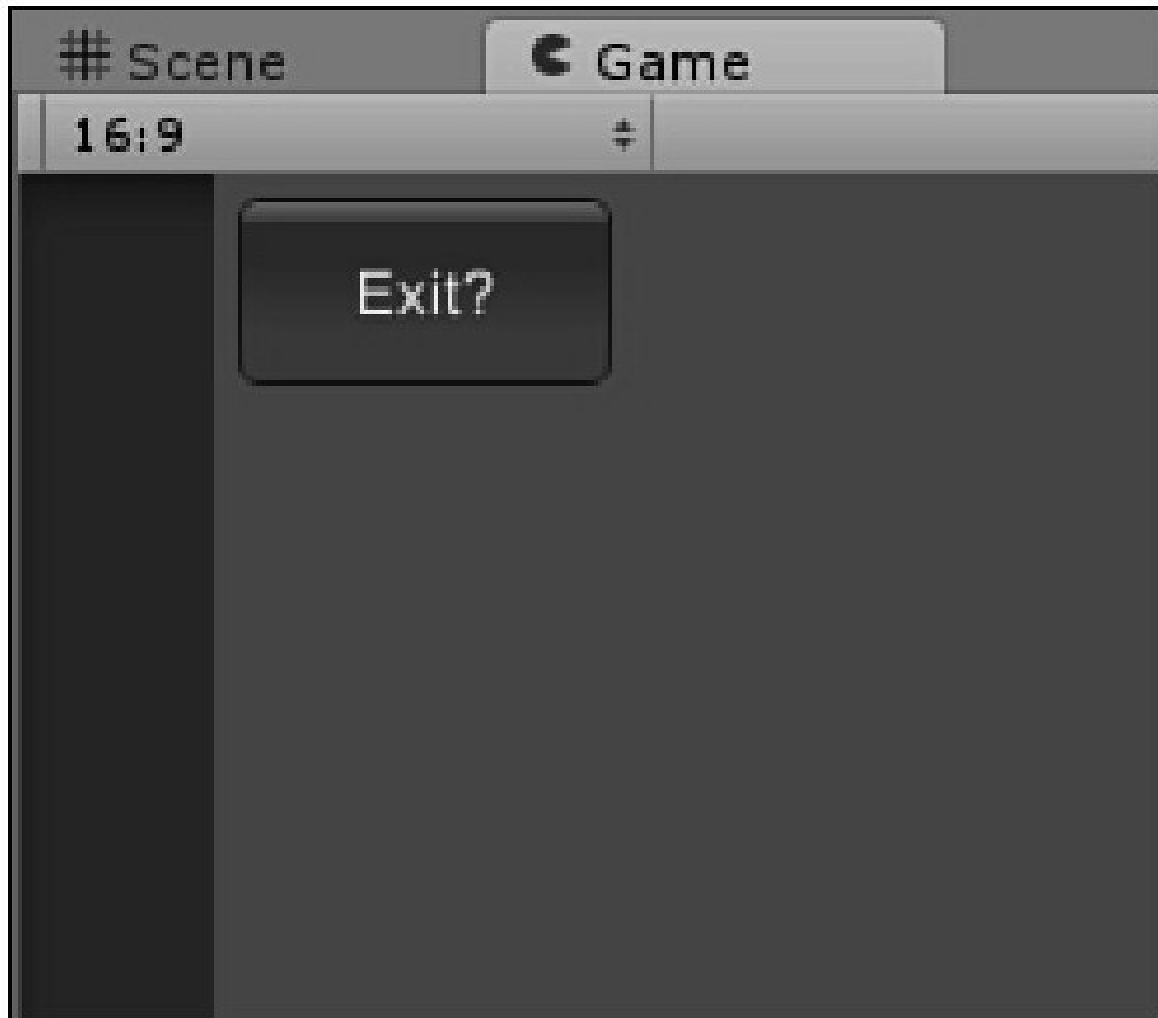


图13.3 按钮控件

13.2.4 重复按钮

重复按钮几乎与按钮完全相同，只不过可以按下并按住它。如果想创建一个按钮，在按住它的整个时间都增加某个变量的值，就可以输入以下代码：

```
if(GUI.RepeatButton(new Rect(0, 0, 80, 20), "Increase"))
{
  someValue += 1;
}
```

同样，仅仅出于演示这个示例的目的添加了someValue变量。

13.2.5 切换开关

切换开关（toggle）也称为状态（stated）按钮，这意味着按钮将保持一种被单击或者未单击的状态（可以考虑开关）。用于切换开关的代码与其他按钮相同，只不过它接受一个布尔参数，并返回一个布尔值。它读入的参数用于确定当前是否单击它，而它返回的布尔值则指示它是否被单击。用于切换开关的语法如下所示：

```
<Some Boolean>= GUI.Toggle(new Rect(<x>, <y>, <w>, <h>), <Some Boolean>, <Some String>);
```

在创建切换开关时，一个好主意是在OnGUI( )方法外面创建一个布尔变量，用于存储切换开关的状态。要创建切换开关，可以编写如下代码：

```
bool toggleState = false;
void OnGUI()
{
   toggleState = GUI.Toggle(new Rect(5, 5, 80, 30), toggleState, "My Toggle");
   }
```

图13.4显示了一个切换开关按钮。



图13.4 切换开关控件

13.2.6 工具栏

工具栏是一排按钮，它所包含的按钮数量取决于你自己。就像正常的工具栏一样，在工具栏上一次只能选择一个按钮，并且使用一个整型变量来跟踪当前选择了哪个按钮。关于工具栏的另一件新事物是字符串数组的使用，数组中的多少项将确定会有多少个按钮出现在工具栏中。用于工具栏控件的语法如下所示：

<Some int>= GUI.Toolbar(new Rect(<x>, <y>, <w>, <h>), <Some Int>, <Array>);

因此，如果想要创建一个工具栏，其中包含几个分别显

示“Easy”、“Medium”和“Hard”的按钮，就可以编写以下代码：

```
int buttonInt = 0;
string[] list = {"Easy", "Medium", "Hard"};
void OnGUI()
{
   buttonInt = GUI.Toolbar(new Rect(5, 5, 200, 30), buttonInt, list);
}
```

工具栏

让我们花点时间试验一下Unity中的工具栏。

（1）创建一个新的项目或场景。然后创建一个名为 GUIScript 的脚本，并把它附加到Main Camera 上。

（2）把前面的工具栏代码添加到脚本中。一定要把代码放在任何方法的外面，但是要放在类里面。

（3）运行场景。应该会看到3个按钮，如图13.5所示，尝试单击按钮，并查看它们是如何交互的。

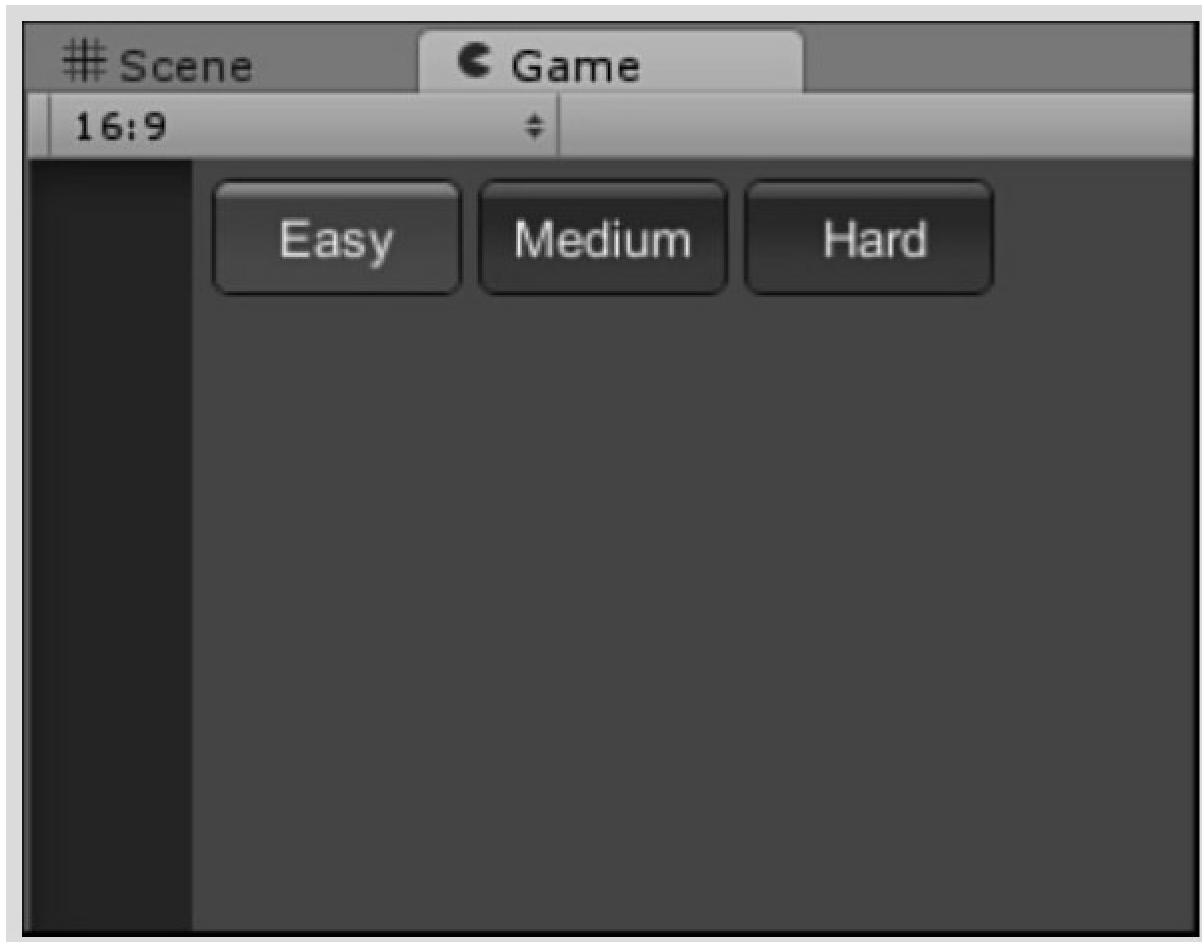


图13.5 工具栏控件

13.2.7 文本框

文本框控件允许用户向场景中输入文本。这个控件本身将显示为一个方框，可以选取它并在其中输入文本，也可以选择把一个字符串放入方框中。就像以前的控件一样，需要传入一个字符串以及从文本框中接受一个字符串，以跟踪用户的交互。用于文本框的语法如下所示：

<Some String>= GUI.TextField(new Rect(<x>, <y>, <w>, <h>), <Some String>);

因此，要创建一个显示“Enter Text Here”的文本框，可以编写以下代码：

```
string textString = "Enter Text Here";
void OnGUI()
{
   textString = GUI.TextField(new Rect(5, 5, 100, 30), textString);
}
```

试一试！要注意的一件事情是，无论文本框有多高，它都只能包含一行文本。图13.6显示了一个文本框。

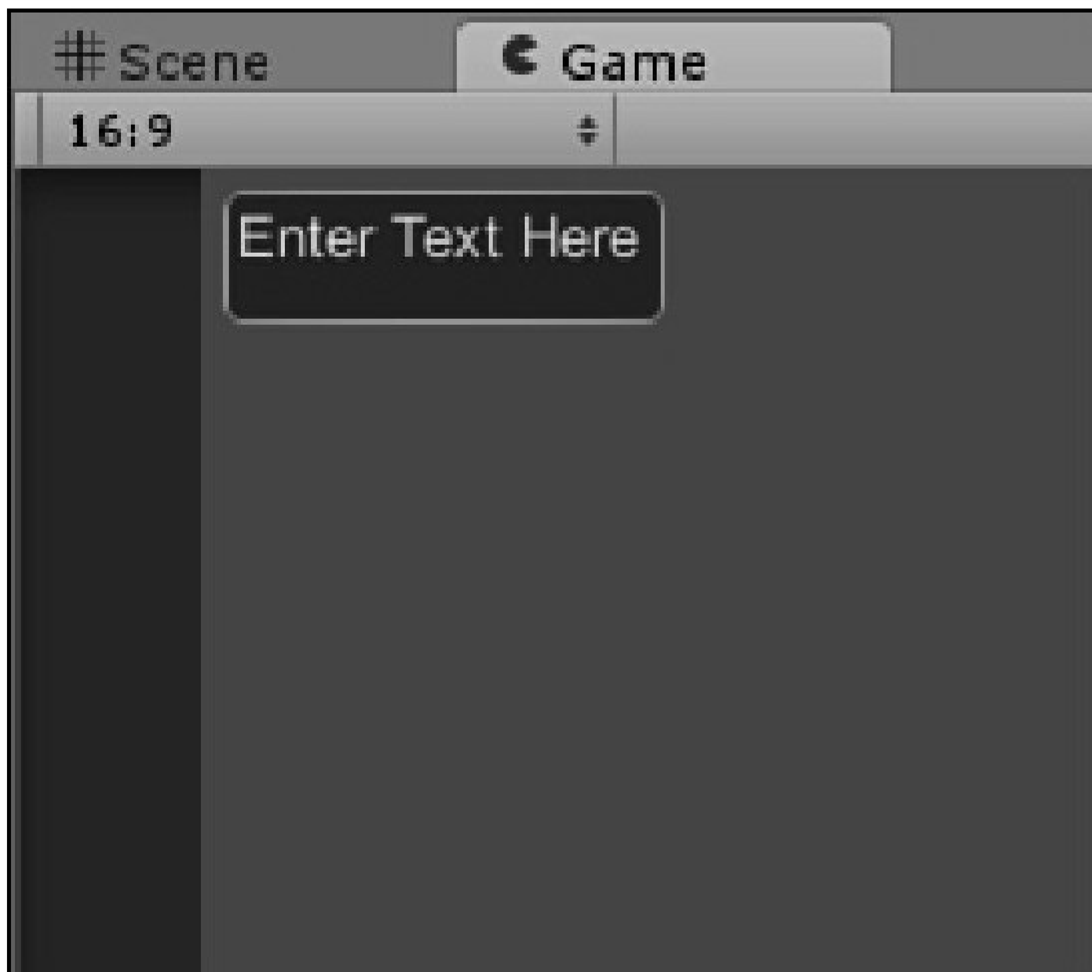


图13.6 文本框控件

13.2.8 文本区

文本区完全就像文本框一样，只不过它可以包含多行文本。用于创

建文本区的语法如下：

```
<Some String> = GUI.TextArea(new Rect(<x>, <y>, <w>, <h>), <Some String>);
```

注意：

由于文本区可以包含多行文本，因此用户有可能输入如此多的文本行，使得文本超出了文本区的垂直空间。

13.2.9 滑块

滑块允许用户通过“滑动”控件，在一系列值当中做出选择。在Unity　中有两类滑块：水平滑块和垂直滑块。除了位置Rect变量之外，滑块还需要3个参数。滑块将读入一个浮点数，指示滑块的当前值。滑块还会读入另外两个参数，分别用于指示最小和最大滑块值。滑块将返回一个浮点数，它包含滑块的值。用于两个滑块的语法如下所示：

```
<Value>= GUI.HorizontalSlider(new Rect(<x>, <y>, <w>, <h>), <Value>, <Min>, <Max>);
<Value>= GUI.HorizontalSlider(new Rect(<x>, <y>, <w>, <h>), <Value>, <Min>, <Max>);
```

因此，要创建两个滑块，其中每个滑块的范围都是0～100，可以编写以下代码：

```
float hValue = 0;
float vValue = 0;
void OnGUI()
{
  vValue = GUI.VerticalSlider(new Rect(5, 5, 20, 150), vValue, 0, 100);
  hValue = GUI.HorizontalSlider(new Rect(30, 30, 150, 20), hValue, 0,
```

```
    100);
}
```

图13.7显示了通过前面的代码创建的两个滑块控件。

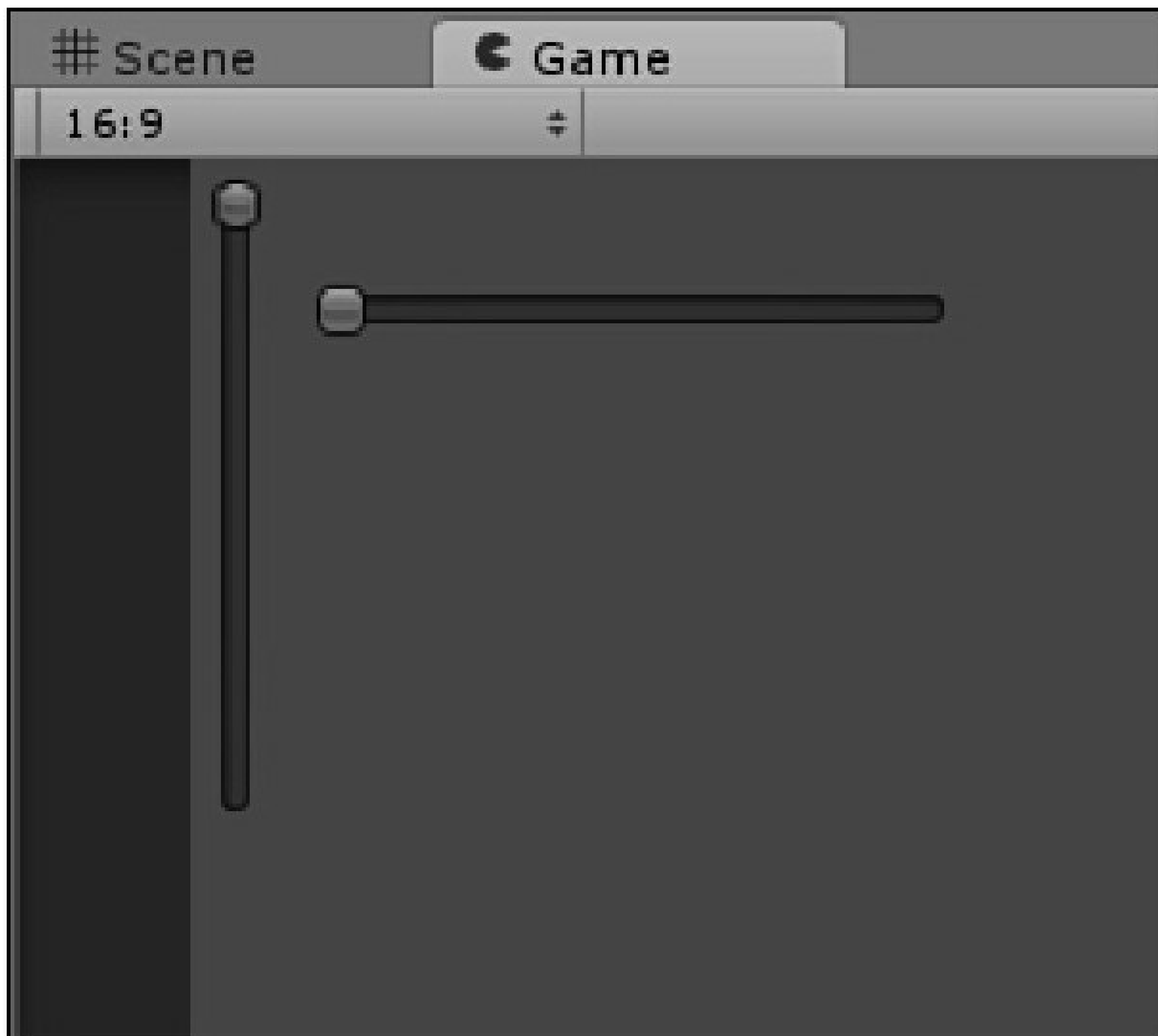


图13.7 滑块控件

13.3 自定义

GUI是任何游戏的一个重要且占据主导地位的部分。Unity的内置GUI系统非常强大，但是你通常想要一种更加自定义的外观和感觉。幸运的是，自定义GUI控件的工作方式是一个简单的过程。可以使用GUI样式和GUI皮肤更改控件。

13.3.1 GUI样式

可以给控件添加一种GUI样式，来指示它如何工作。在Unity中构建了这些样式，用于模拟Web 页面中使用的CSS（Cascading Style Sheets，层叠样式表），并且允许更改文本颜色、背景纹理、字体等。

每个 GUI 控件都已经应用了一种默认的 GUI 样式，并且样式的名称与用于控件的名称相同。例如，一个按钮就具有一种应用于它的名为按钮的样式。当你意识到可以把一个控件的样式应用于另一种类型的控件时，这将变得很有趣。例如，如果要把按钮样式应用于切换开关，将获得一个外观像按钮但是行为方式像切换开关的控件。本章前面讨论的每种控件都具有提供额外一个参数的选项，这个参数是样式参数，它可以是一个GUIStyle对象，或者是带有样式名称的字符串。

混合与匹配样式

在这个练习中，将创建一个看起来像按钮的切换开关。

（1）创建一个新的项目或场景。向场景中添加一个名为 GUIScript 的脚本，并把它附加到Main Camera 上。

（2）向脚本中添加以下代码。注意切换开关具有一个额外的参数，它刚好就是名称“button”：

```
bool value = false;
void OnGUI()
{
    value = GUI.Toggle(new Rect(5, 5, 100, 100), value, "toggle",
    "button");
}
```

（3）运行场景，并且注意切换开关看上去像按钮一样。当你单击它时，注意所发生的事情。继续前进，并试验不同的控件和样式。记住：控件样式只是控件名称。

如果你不想重用内置的控件样式之一，可以创建你自己的控件样式。有一种方式在代码中构建样式，但是使用编辑器要容易得多。要使用编辑器构建样式，首先必须向脚本中添加一个GUIStyle变量。下面详细描述了这样做的步骤。

（1）向场景中添加一个脚本，其中带有一个OnGUI()方法（如果还没有该方法的话）。如果你具有该方法，将希望使用它。确保把这个脚本附加到一个对象上。

（2）向脚本中添加一个GUIStyle变量，用于执行该操作的语法如下：

```
public GUIStyle <variable name>;
```

这段代码出现在类里面，但是位于任何方法的外面。

（3）在Unity中，选取附加了脚本的对象，并且注意脚本组件上的Style属性，如图13.8所示。可以单击样式的不同属性，并按你所喜欢的那样更改它们。



图13.8 Style属性

创建自定义的样式

在这个练习中，将创建一种自定义的样式，并将其应用于自定义的控件。

（1）创建一个新的项目或场景。然后添加一个名为 GUIScript 的脚本，并将其附加到摄像机上。

（2）把以下代码添加到脚本中：

```
public GUIStyle style;
```

```
void OnGUI()
{
   if(GUI.Button(new Rect(5, 5, 100, 30),"Hello World", style);
}
```

（3）在Unity中，在Inspector 视图中展开Style 和Normal 属性，并把Text Color属性改为橙色，如图13.9所示。运行场景，并且观察标签的外观。

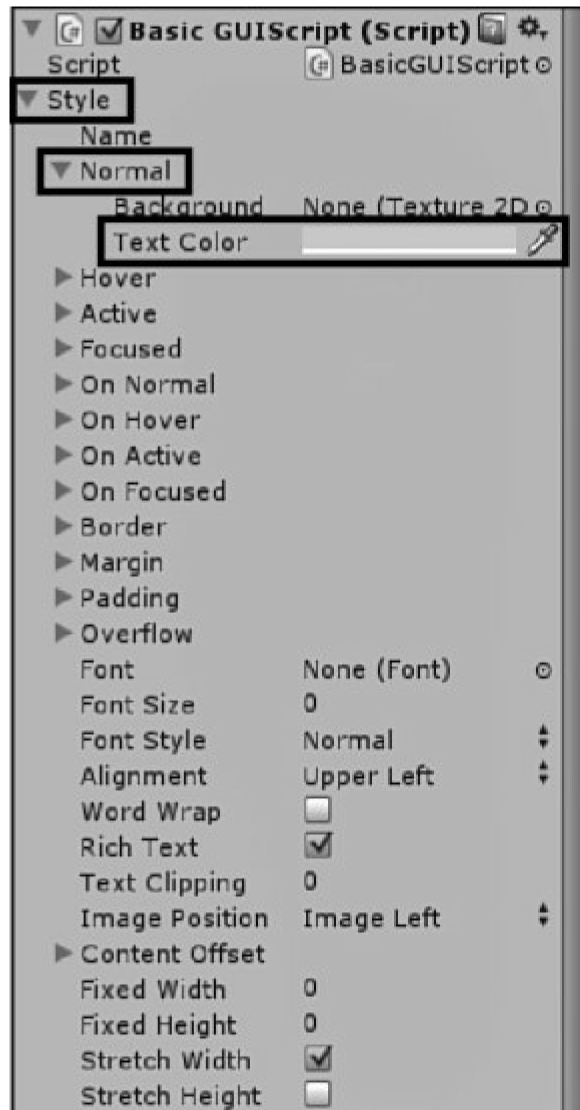


图13.9 更改正常的文本颜色

警告：

样式复杂性

在试验样式时，你可能注意到一些特性没有起到任何作用，还可能注意到给按钮（或者其他任何控件）应用一种样式将使它看起来就像标签一样。这是由于样式具有许多复杂性。例如，使按钮看起来像按钮的图形就是：一幅图形。如果没有在样式中提供该图形，按钮看起来将不像按钮。对于被单击的按钮也是如此。按下的按钮图像事实上是另一幅图像。因此，如果计划为控件创建自己的样式，就要花一些时间考虑所需的所有资源，以使控件看上去像你想要的那样。

13.3.2 GUI皮肤

GUI样式指定控件在渲染时的外观。如果只需要管理几个控件，这很不错。不过，如果需要为带有许多不同控件的整个 GUI 构建“外观和感觉”，就可能难以维护所有需要的不同样式。此时，GUI皮肤就可以派上用场了。实质上，GUI皮肤只是样式的集合。通过创建一种皮肤，将能够为项目的所有不同的控件指定外观。

要给项目添加一种皮肤，只需在Project视图中右键单击一个文件夹，并选择Create > GUI Skin命令。选择新创建的皮肤将在Inspector视图中显示一个样式列表，这些是用于每个GUI控件的样式。还有另外几个控件可用，比如用于所有控件的通用字体。把皮肤链接到GUI是在脚本中处理的。需要在脚本中创建一个GUISkin变量，其语法如下：

```
public GUISkin <variable name>;
```

在编辑器中给变量提供一个值之后，只需把它指定给GUI即可。用于所有这些操作的语法如下：

```
public GUISkin skin;
void OnGUI()
{
```

```
    GUI.skin = skin;
    //GUI code goes here
}
```

处理GUI皮肤

让我们试验一下GUI皮肤。

（1）创建一个新的项目或场景。然后添加一个名为 GUIScript 的脚本，并把它附加到摄像机上。

（2）向脚本中添加以下代码：

```
public GUISkin skin;
void OnGUI()
{
    GUI.skin = skin;
    if(GUI.Button(new Rect(5, 5, 100, 30), "Hello World"))
    {}
}
```

（3）在Unity中，创建一种GUI 皮肤（右键单击Assets 文件夹，并选择Create > GUI Skin命令），并把它命名为NewSkin。展开 Button 属性，然后展开Active 属性，并把Text Color属性改为红色，如图13.10所示。

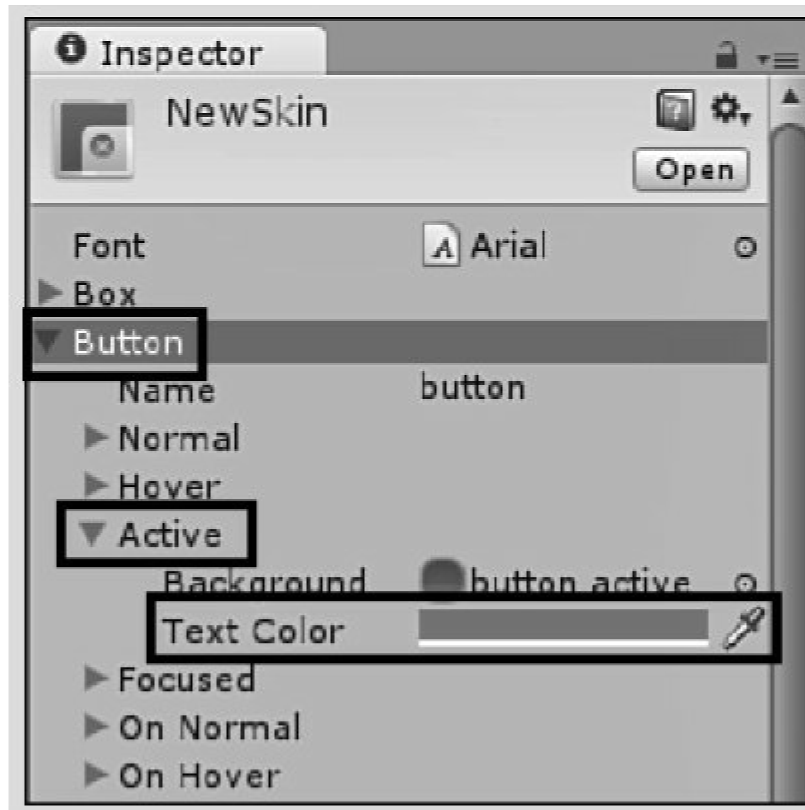


图13.10 更改活动的文本颜色

（4）选取摄像机，单击并把 NewSkin 资源拖到 GUI Script 组件的 Skin　属性上，如图13.11所示。运行场景并单击按钮，注意文本颜色怎样改为红色。

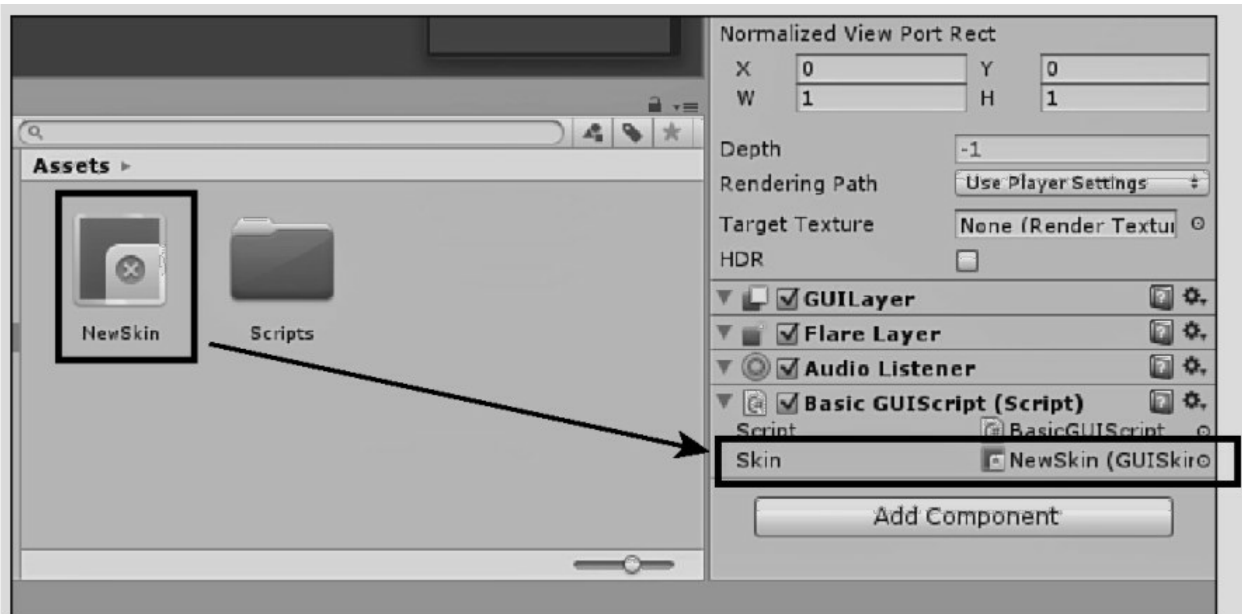


图13.11 应用GUI皮肤

注意：

关于字体

可以同时使用样式和皮肤指定GUI控件的字体。Unity中的字体就像任何其他的资源一样，只需把想要的字体拖到Assets文件夹中，Unity将自动识别它，然后可以简单地把它应用于任何字体属性。字体的唯一要求是它们必须是.ttf或.dfont文件类型。

13.4 小结

在本章中，你学习了Unity中的GUI的相关知识。你首先学习了GUI的基础知识，以及如何设计和创建它们。接着，你学习了定位 GUI 控件，接着研究了许多常用的 GUI 控件并且试验了它们。在本章最后，你学习了有关GUI样式和皮肤的知识。

13.5 问与答

问：每一款游戏都需要GUI吗？

答：通常，游戏都会受益于经过深思熟虑而开发的GUI。完全没有GUI的游戏是极其少见的。也就是说，使用GUI总是一个好主意。你肯定不希望由于游戏过于混乱而加重玩家的负担。

问：使用GUI有任何性能考虑吗？

答：是的。Unity中的 GUI系统如果使用过度，那它可能就是一个效率非常低的系统，在移动平台上尤其如此。这并不是说不应该使用GUI系统，而只应该节俭地、适当地使用它。同样，只应该使用 GUI 显示所需的信息，而不要尝试把太多的内容放在屏幕上。

13.6 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

13.6.1 问题

1．GUI意指什么？

2．哪种变量类型存储x和y位置以及宽度和高度？

3．当说道切换开关是一个状态按钮时，它意味着什么？

4．GUI样式与GUI皮肤之间的区别是什么？

13.6.2 答案

1．图形用户界面。

2．Rect。

3．切换开关将维持一种状态。也就是说，切换开关知道它是否被单击。

4．样式指定了一个控件的外观；皮肤是样式的集合，用于给整个GUI 系统提供一致的外观和感觉。

13.6.3 练习

在这个练习中，你设计了自己的 GUI 系统。出于创造性考虑，将允许你按自己想要的任何方式编排 GUI 的样式。在用于第13章（Hour 13）的本书配套资源中完成的练习示例（名为 Hour13_Exercise）只使用了默认的控件样式。这个练习将要求你给它提供一种独特的样式。项目本身是一个简单的程序，看看你是否可以自己弄明白它。如果你有困

难，一定要检查示例。

向场景中添加一个文本框，它应该不包含任何文本。

向场景中添加一个按钮，它应该显示“Click Me”。

向场景中添加一个标签，它应该不包含任何文本。

当单击按钮时，从文本框中获取文本，并把它放在标签中。

使用GUI 皮肤，给文本框、按钮和标签提供一种独特的外观。它们应该具有一致的颜色模式和字体。在这里可以自由地发挥你的创造性，这是你展示独特个性的机会。

第14章 角色控制器

在本章中你将学到：

Unity的角色控制器的基础知识；

怎样为角色控制器创建脚本；

怎样构建一个简单自定义的角色控制器。

在本章中，你将学习Unity中的角色控制器组件。你首先将学习角色控制器的基础知识，了解它是什么以及它是如何工作的。接着，你将学习如何编写脚本来操纵和利用角色控制器的能力。在本章最后，你将从头开始构建自己的角色控制器。

注意：

为什么要学习关于控制器的知识？

你可能感到迷惑的是：当 Unity 已经提供了两种非常健壮的控制器（First Person控制器和Third Person控制器）时，为什么还需要学习关于角色控制器的知识。事实上，尽管这两种控制器非常强大，但是在许多情况下它们并不能够满足需要。如果你想为2D游戏创建控制器，该怎么办？如果你希望控制器以不同的方式计算重力，又该如何？理解控制器的基本工作方式很重要，这样就可以为项目构建特定的解决方案。

14.1 角色控制器

迄今为止，你已经见过了许多种与场景中的对象交互的方式。你探索了通过脚本手动移动它们的方式，还见过了利用刚体进行的物理交互。正常情况下，这些是在游戏中获得运动的可接受的方式。不过，如果你正在寻找更现实、更一致的游戏玩法，将需要更强大一点的功能，这就是角色控制器，通常简称为控制器（controller）。角色控制器是一种专用组件，允许在不利用刚体物理学的情况下对游戏对象进行高级控制。也就是说，角色控制器允许沿着地面移动对象，并且会受到墙壁和陡峭的山丘约束，而不会挤压对象或者被对象挤压。本质上讲，角色控制器就是一个具有一些额外功能的胶囊碰撞器，本章后面将探讨这些功能。

14.1.1 添加角色控制器

角色控制器本身是一个组件，可以应用于任何游戏对象。尽管角色（character）这个术语一般暗指玩家，但它可用于控制场景中所有移动的实体（玩家、敌人、汽车等）。要把角色控制器应用于某个对象，只需选取该对象，然后单击 Component > Physics > Character Controller命令即可。角色控制器现在应该出现在Inspector视图中的对象下面。

注意：

关于碰撞器的争论

由于角色控制器具有它自己的胶囊碰撞器，在尝试把它添加给已经具有碰撞器的对象时，可能会接收到一条警告消息，如图14.1所示。你可以选择取消角色控制器，利用新的胶囊碰撞器替换现在的碰撞器，或

者把它们二者都保留在对象上。你想具有的效果将确定你将选择哪个选项。一般来讲，如果把角色控制器用于正常的移动，将希望利用胶囊碰撞器替换现有的碰撞器。



图14.1 在把控制器添加给带有碰撞器的对象时出现的消息

给对象添加角色控制器

在这个练习中，将给场景中的对象添加角色控制器。

（1）创建一个新的项目或场景，并向场景中添加一个立方体。

（2）选取这个立方体，并且单击Component > Physics > Character Controller 命令，添加一个角色控制器。

（3）立方体现在应该在Inspector视图中具有角色控制器组件。此外，它可能比较暗淡，但是应该能够看到胶囊碰撞器出现在Scene视图中的立方体里面。

警告：

角色控制器和刚体

由于角色控制器和刚体都是组件，可以把它们都添加到同一个对象上。不过，这不是一个好主意。控制器和刚体都将尝试以它们自己特定的方式控制对象的运动，这将在对象中导致奇怪的行为。一般规则是只选择其中之一，仅当尝试实现特定的目标并且知道自己正在做什么时，

才可以同时使用它们二者。

14.1.2 角色控制器的属性

角色控制器具有两组属性。有些属性是在Unity编辑器中通过Inspector视图使用的（将在这里介绍），有些属性则是通过脚本访问的（将在后面介绍）。图14.2显示了角色控制器组件的不同属性。

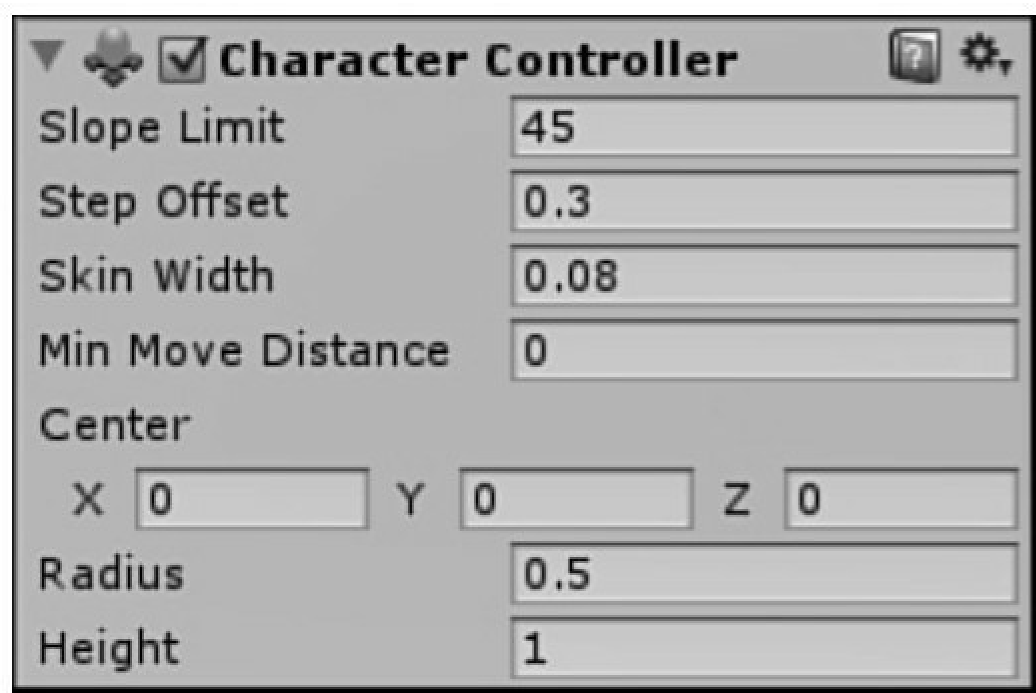


图14.2 角色控制器组件

表14.1描述了角色控制器的所有属性。

表14.1 角色控制器的属性

属性	描述
Slope Limit	确定控制器可以爬上的最陡的斜坡。控制器将无法逾越比这里指定的值更陡的斜坡
Step Offset	比指定的值更接近地面的任何台阶都将是可以爬上的；比指定的值更高的台阶都将无法逾越
Skin Width	确定在检测到碰撞之前一个碰撞器可以刺入控制器的碰撞器有多深。宽度太小将导致控制器抖动；宽度太大将导致控制器被卡住。一种良好的一般设置是控制器半径的 10%
Min Move Distance	确定可以指示控制器移动的最短距离。这可用于减小抖动，但把它设置得太高可能导致控制器感觉反应迟钝。一条良好的规则是把它设置为 0，并且仅当需要时才增加它
Center	属于角色控制器的胶囊碰撞器的中心
Radius	属于角色控制器的胶囊碰撞器的半径
Height	属于角色控制器的胶囊碰撞器的高度

14.2 用于角色控制器的脚本

你在前面可能注意到，只是简单地把控制器放在游戏对象上并不能起到多大的作用。事实上，如果创建了一个具有任何下落或移动项目的场景，将会看到它们与包含控制器的对象发生碰撞，但是控制器将不会被它们移动。角色控制器的大部分能力都存在于脚本中。注意：角色控制器只是简单地提供控制的基础，实际的实现则完全取决于你自己。这意味着必须做更多一点的工作来创建控制器，但是结果可能更强大、更量身定制并且更精细。

14.2.1 控制器脚本编程

如本章前面所述，角色控制器具有一系列可以通过脚本访问的属性（变量），它们给角色控制器提供了许多能力。不过，在可以利用代码处理控制器之前，必须获得一个指向它的引用：

```
CharacterController controller;
void Start (){
    controller = GetComponent<CharacterController>();
}
```

这段代码将创建一个 CharacterController 变量。然后，在 Start( )方法中，它将找到控制器引用，并把它保存到变量中。这样，将能够在代码中使用它。表14.2描述了角色控制器脚本编程变量。

注意：

常用功能

角色控制器组件是碰撞器组件的后代。我们称角色控制器“继承”自

碰撞器。因此，控制器能够访问也属于碰撞器的所有脚本能力。不过，本小节将只介绍角色控制器所特有的项目。值得指出的是控制器与碰撞器之间的关系，以免你在代码中注意到一些额外的功能，并且想知道它们来自哪里。

表14.2 角色控制器脚本编程变量

属性	描述
slopeLimit、stepOffset、center、radius、height	这些与表 14.1 中提到的组件属性相同
isGrounded	它是一个布尔值，指示控制器当前是否接触到地面，可用于跳跃和飞行力学
velocity	一个 Vector3 变量，指示控制器沿着每根轴行进得有多快。这是通过计算 Move( )或SimpleMove()调用之前和之后的位置确定的（将在以后讨论）
collisionFlags	一个 CollisionFlags 变量，指示碰撞发生在控制器上的什么位置（将在后面更详细地介绍）
detectCollisions	一个布尔变量，确定角色控制器是否将检测与其他碰撞器和刚体发生的碰撞。默认情况下，将把它设置为 true

除了一组变量之外，角色控制器还提供了两个新方法：SimpleMove( )和Move()。这些方法使用运动的思想四处移动对象。这意味着不会放置对象，也不会推动它们。它们也不会平移。其效果将基于在Input Manager中设置的实际输入控制，结果就是控制器的移动会一点一点地累积或减弱。这使得它感觉更逼真。

bool SimpleMove(Vector3 movement)

顾名思义，SimpleMove( )是四处移动对象的简单方式。这个方法读入一个Vector3 变量，它包含对象应该沿着每根轴移动多远的距离。该方法返回一个布尔值，指示对象是否放在地上（接触地面）。在内部，该方法将自动给对象应用重力。因此，SimpleMove()方法将忽略你提供的在y轴中的任何移动。结果就是你不能使用自己的重力，也不能利用SimpleMove()应用任何跳跃或飞行。如果你想要这些，将需要代之以使用Move()。要注意的另外一件事是：SimpleMove()将以不同于 Move()的方式移动距离。因此，要行进相同的距离，利用SimpleMove()移动对象的距离将不同于利用Move( )移动对象的距离：

CollisionFlags Move(Vector3 movement)

像SimpleMove()一样，Move()负责在场景中四处移动对象。它接受一个Vector3 变量，该变量包含你希望沿着每根轴移动的距离。Move()返回一个CollisionFlags变量（将在后面介绍），该变量包含在移动期间发生的任何碰撞。Move()方法不会应用任何重力，因此你必须自己计算并应用它。

提示：

控制滑动

利用角色控制器进行的移动可能包含某种滑动（slide）。也就是说，当按下某个键时，对象不会立即停止。作为替代，对象将随着时间的推移放慢速度，直至停下来。可以通过在Input Manager 中更改输入轴的Gravity属性，增加或减小对象所具有的滑动距离，如图14.3所示。要打开 Input Manager，可以单击Edit > Project Settings > Input 命令。

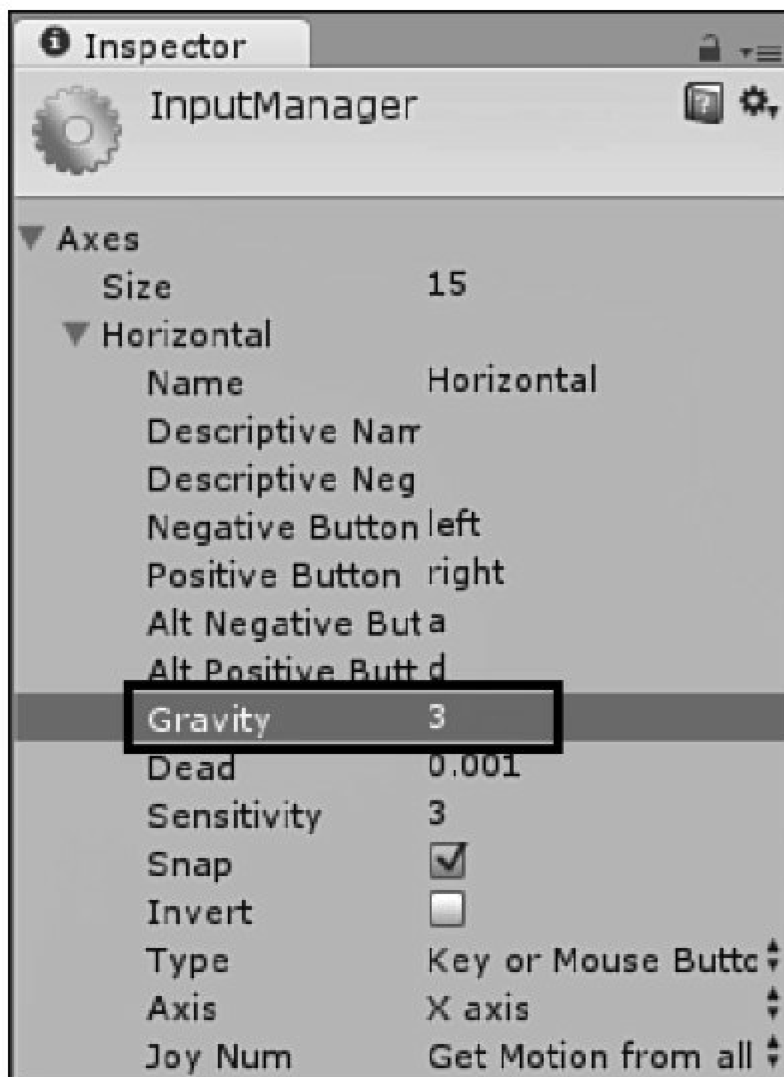


图14.3 Input Manager 的重力设置

14.2.2 CollisionFlags

CollisionFlags变量类型是一个复杂的变量，其中包含关于角色控制器所发生的碰撞的信息。该变量是一个位掩码（bitmask），这意味着数据存储在二进制代码自身内。所有这些对你来说意味着有一种不同的方式可以从中提取你需要的信息。CollisionFlags 变量可以是一个值，或者包含一个值。它们之间的区别是：如果标志是某个值，就会排除所有其他的值。与之相反，CollisionFlags 变量也可以包含许多不同的值。利用

一个示例来说明这些将更有意义。

假设你想确定某个对象是否只会与它下面的对象发生碰撞，可以编写以下代码：

```
if (controller.collisionFlags == CollisionFlags.Below)
   print("This is only colliding with an object below");
```

如果上面的条件为“真”，你就知道该对象不能在任何其他的方向上发生碰撞。如果你想确定对象将沿着底部发生碰撞，但是也能在另一个方向上发生碰撞，就可以编写以下代码：

```
if (controller.collisionFlags & CollisionFlags.Below)
   print("This is colliding with an object below. Could be colliding elsewhere.");
```

它们二者之间的区别是：在第一个代码示例中，值是 Below；而在第二个代码示例中，它只是简单地包含Below。显然，CollisionFlags变量只能等于None。它不能同时包含None值和另一个值。既要使一个碰撞器不发生碰撞，同时仍然又使其在某个方向上发生碰撞，这显然是不可能实现的。CollisionFlags变量类型可以包含None、Sides、Above或Below这些值，它们可以写成如下形式：

```
CollisionFlags.None
CollisionFlags.Above
CollisionFlags.Sides
CollisionFlags.Below
```

使用这些标志，可以精确确定一个对象怎样与控制器发生碰撞。

14.2.3 碰撞

角色控制器在移动时会自动处理碰撞，但是有时你希望进行更精细的控制。这就是无论何时检测到碰撞控制器都会调用

OnControllerColliderHit()方法的原因。使用该方法，可以编写你自定义的碰撞效果（比如推动对象）。要检测碰撞，将需要向脚本中添加以下代码：

```
void OnControllerColliderHit(ControllerColliderHit hit）{
    //your collision code goes here
}
```

在把该方法添加到代码中之后，就可以把你想要的任何碰撞代码放入其中。参数hit将包含关于同控制器发生碰撞的对象的信息。在本章后面将给出关于该方法的实用的探讨。

14.3 构建控制器

既然你已经学习了Character　Controller组件，并且查看了如何处理它，现在就可以开始构建你自己的控制器。注意：没有哪两个控制器完全相同。在设计它们时，就使它们能够轻松地进行自定义，以便精确地满足你的需要。因此，本书这一节中介绍的并不是创建控制器的通用方式，而只是创建控制器的一种方式。

本书中在这里可以展示许多不同的控制器。你将创建的控制器类型打算用于像 Super Mario Bros 这样的2D平台型游戏。在用于第14章的本书配套资源中可以找到完整的项目和控制器脚本，其名称为Hour14_Controller。

14.3.1 初始设置

在实际地研究自定义控制器的脚本编程之前，你将希望建立一个场景，以试验不同的方面。这个场景将足够简单，包含地面、单一平台以及一个四处移动的角色。

（1）创建一个新的项目或场景，并添加一个定向灯光、两个立方体和一个胶囊。

（2）由于这是一个2D场景，你将希望正确地设置摄像机。选取摄像机，在Inspector视图中把Projection属性改为Orthographic，并把Size属性改为8，如图 14.4所示。然后把摄像机的位置改为(0, 2.4, −10)。



图14.4 摄像机属性

（3）把其中一个立方体放在(0, 0,−5)处，并将其缩放为(20, .5, 2)，它将充当地板。把另一个立方体放在(3, 3, −5)处，并将其缩放为(3, .5, 1)。

（4）把胶囊放在(0, 2,−5)处，并给它添加一个角色控制器（单击Component > Physics >Character Controller 命令）。在提示时，继续前进，并利用新的碰撞器替换现有的碰撞器。

（5）向场景中添加一个Scripts文件夹，然后向该文件夹中添加一个名为ControllerScript的脚本，并把该脚本附加到胶囊上。这将是由角色控制器控制的对象。修改控制器脚本，以包含下面的代码：

```
CharacterController controller;
void Start (){
    controller = GetComponent<CharacterController>();
}
```

如果运行场景，将会看到类似于如图14.5所示的内容。既然已经建立了场景，就可以开始处理多个不同的功能。

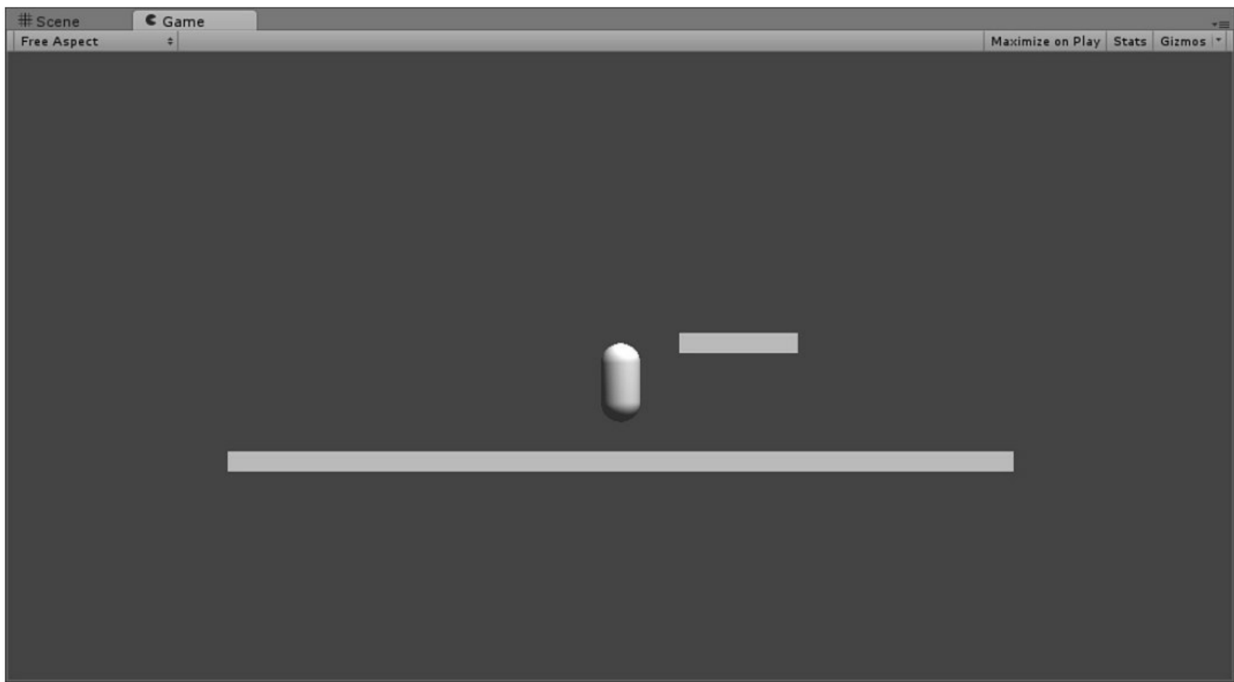


图14.5 完成的场景

14.3.2 运动

既然已经完成了场景，你将希望给胶囊添加最基本的功能：运动。要移动对象，将希望计算运动矢量，并调用Move(　)方法。把想要的运动速度存储在一个变量中以便可以轻松地修改它，这也是一个好主意（但不是必需的）。最后，你将希望把运动信息存储在一个　Vector3变量中。这仍然不是必需的，但是在后面当开始使用y轴时将需要它：

```
public float speed = 5.0f;
```

```
Vector3 movement = Vector3.zero;
void Update (){
   movement.x = Input.GetAxis("Horizontal"）* speed;
   controller.Move(movement * Time.deltaTime);
}
```

上面的代码首先声明一个速度变量，它将用于控制对象移动的速度。接下来，它声明一个名为movement的Vector3变量，它将用于存储从一个帧到另一个帧的运动信息。然后，在Update( )方法内，将把水平运动轴的值（左/右箭头或A/S键）与速度的乘积赋予名为movement的变量。注意：这里没有给出y轴或z轴的值，因为这是一个2D控制器，因此没有z轴方向的运动，并且y 轴是以不同的方式处理的。最后，调用Move( )方法，并且把movement 变量与Time.deltaTime相乘。执行这个乘法后，就可以确保无论帧速率是多少，场景在任何计算机上都将以完全相同的方式运行。

运行场景，并且注意现在可以怎样来回移动胶囊。你应该会注意到胶囊将被平台阻止，还会注意到胶囊只是飘浮在空气中。这是由于（还）没有应用重力。

14.3.3 重力

接下来你将希望给控制器添加重力。处理重力的方式有两种：可以为重力使用内置的值，或者可以指定你自己的值。为重力应用内置的值将使所有的物体都以相同的速率下落。不过，如果你希望某个角色以不同的方式下落（如使用降落伞），将需要使用你自己的值。可以通过添加以下代码来应用重力：

```
movement.x = Input.GetAxis("Horizontal"）* speed;
if(controller.isGrounded == false)
```

```
    movement.y += Physics.gravity.y * Time.deltaTime;
controller.Move(movement * Time.deltaTime);
```

这段代码中的第一行和最后一行前面已经介绍过，这里保留了它们是为了便于参考。由于重力并不总是需要应用，因此使用一条if语句来确定角色是否没有接触地面。如果确定角色当前没有碰撞底面，就会将场景的当前重力的y　成分应用于运动矢量。这样，在调用Move(　)方法时，对象将左右移动，但它也会受到重力影响。

运行场景，看看它的实际状况。你将注意到胶囊会立即下落，并在撞到地面时停止。如果使胶囊从平台的一侧落下，就可以看到它离开了场景。

14.3.4 跳跃

如果不能从一个平台跳跃到另一个平台，那么平台游戏将不是非常有趣。跳跃比移动和下落更复杂一点，需要跟踪角色可以跳得多高。你还希望确保角色每次只跳跃一下，否则它们将会飞起来：

```
public float jumpSpeed = 8.0f;
  void Update(){
    //movement and gravity code
    if (Input.GetButton("Jump"）&& controller.isGrounded == true)
      movement.y = jumpSpeed;
  controller.Move(movement * Time.deltaTime);
}
```

同样，这里给出了更多已经介绍过的代码，把它们放在这里只是为了便于参考。第一段代码声明一个新的浮点型变量jumpSpeed，它指定角色可以跳多高。然后，在Update(　)方法内，使用一条if语句确保仅当按下键并且仅当角色当前位于地面上时，它才能够跳跃。

运行场景并试试它，看看是否可以使胶囊跳到第二个平台上。注意当胶囊在空中飞行时怎样控制它。这是一个特定的设计选择，如果需要可以在将来的项目中修改它。

14.3.5 推动对象

你想添加的最后一点功能是在场景中四处推动对象的能力。这将需要前面提过的OnControllerColliderHit( )方法，用于该功能的代码如下所示：

```
public float pushPower = 2.0f;
  void OnControllerColliderHit(ControllerColliderHit hit）{
    Rigidbody body = hit.collider.attachedRigidbody;
    if (body == null || body.isKinematic)
      return;
    if (hit.moveDirection.y < -0.3f)
      return;
    Vector3 pushDir = new Vector3(hit.moveDirection.x, 0f, 0f);
    body.velocity = pushDir * pushPower;
}
```

一定要把这段代码添加到类中，但是要使它们位于任何其他的方法之外。第一行代码创建一个变量，用于确定控制器将可以多么用力地推动另一个对象。然后，在OnControllerColliderHit( )方法内，你将具有“推动”代码。在该方法中，将获得一个指向被碰撞对象的刚体的引用。如果刚体不存在或者是运动的，方法就会退出。从此，可以检查碰撞的方向，以确保不会推动控制器下面的对象。一旦执行了所有这些检查，就可以计算推动的方向，然后把推动方向与推力相乘，并把结果赋予刚体对象的速度。

在试验它之前，可以向场景中添加一个球体，并把它定位在(1.5, 1, −5)处。确保给球体也添加一个刚体组件。一旦完成这些操作，就可以运行场景。注意胶囊现在可以怎样四处移动球体。尝试沿着平台来回推动球体。

14.3.6 完整的代码清单

这里提供了控制器脚本的完整代码。出于组织结构方面的考虑，这里重新排列了原始程序清单中的一些代码：

```
using UnityEngine;
using System.Collections;
public class ControllerScript : MonoBehaviour {
   CharacterController controller;
   Vector3 movement = Vector3.zero;
   public float speed = 5.0f;
   public float jumpSpeed = 8.0f;
   public float pushPower = 2.0f;
   void Start () {
      controller = GetComponent<CharacterController>();
   }
   void Update() {
      movement.x = Input.GetAxis("Horizontal") * speed;
      if(controller.isGrounded == false)
         movement.y += Physics.gravity.y * Time.deltaTime;
      if (Input.GetButton("Jump") && controller.isGrounded == true)
         movement.y = jumpSpeed;
      controller.Move(movement * Time.deltaTime);
```

```
    }
    void OnControllerColliderHit(ControllerColliderHit hit) {
        Rigidbody body = hit.collider.attachedRigidbody;
        if (body == null || body.isKinematic)
            return;
        if (hit.moveDirection.y < -0.3f)
            return;
        Vector3 pushDir = new Vector3(hit.moveDirection.x, 0f, 0f);
        body.velocity = pushDir * pushPower;
    }
}
```

14.4 小结

在本章中，你学习了 Unity 的角色控制器，首先研究的是角色控制器的基础知识和组件属性。接着，你学习了通过脚本处理控制器，并且学习了控制器的变量、方法和碰撞。最后，你编写了一个自定义的2D角色控制器，它专门针对平台型游戏。

14.5 问与答

问：角色控制器一共有多少种？

答：只有单独一种角色控制器组件。不过，可以使用它的方式几乎是无限的。在创建角色控制器时，将使你能够针对任何情况量身定制它。

问：在刚体或角色控制器这二者中，哪一种更好用？

答：这是一个重要的问题，答案取决于你想实现什么。如果你指望利用 Unity 的物理功能，那么就可以选择刚体。如果你想自己为角色编写更特定的行为，那么角色控制器就是最重要的。

14.6 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

14.6.1 问题

1．角色控制器提供了什么形状的碰撞器？
2．哪个属性确定在检测到碰撞前碰撞器可以刺入控制器有多深？
3．哪种变量类型包含关于碰撞发生的方向的信息？
4．哪个方法在移动控制器时允许像跳跃这样的y轴运动？

14.6.2 答案

1．胶囊碰撞器。
2．Skin Width 属性。
3．CollisionFlags变量类型。
4．Move( )方法。

14.6.3 练习

这个练习更多地是让你试验编写脚本，而不是其他任何方面。你的挑战是修改本章为你提供的控制器脚本，使其包含以下功能。与以往一样，如果你感到迷茫并且需要帮助，可以在用于第14章（Hour 14）的本书配套资源中找到完整的项目，其名称为Hour14_Exercise。

更改控制器，使得仅当玩家触地时才能更改运动方向。目前，玩家可以在空中更改方向。

允许玩家在按住Shift 键时冲刺（更快地移动）。

允许玩家跳跃两次，它的意思是玩家可以跳跃起来，然后在空中再跳跃一次（此时只能跳跃一次）。

第15章 第3款游戏：Captain Blaster

在本章中你将学到：

怎样设计Captain Blaster游戏；

怎样构建Captain Blaster游戏世界；

怎样构建Captain Blaster实体；

怎样构建Captain Blaster控制；

怎样进一步改进Captain Blaster 游戏。

让我们制作一款游戏！在本章中，将制作一款名称为Captain Blaster的2D滚动射击游戏。你首先将设计游戏的多个元素。接着，将开始构建滚动背景。一旦建立了运动的理念，就可以开始构建多个游戏实体。在完成了实体之后，将构造控制并使项目游戏化。在本章最后，将分析游戏，并确定一些可以改进的地方。

提示：

完成的项目

一定要遵循本章中的指导，来构建完整的游戏项目。如果你感到迷茫，可以在用于第15章（Hour　15）的本书配套资源中找到游戏的完整副本。如果需要帮助或灵感，可以看一看它。

15.1 设计

在第7章中，你已经学习过设计元素是什么。这一次，你将开始接触到它们。

15.1.1 理念

如前所述，Captain Blaster是一款2D滚动射击类游戏。它的前提是玩家将在关卡前飞来飞去，摧毁流星，并设法存活下来。关于 2D 滚动射击类游戏的优点是玩家自身实际上根本不必移动。滚动背景将模拟玩家前进的状态，这减少了玩家必需的技能，并且允许在敌人的形式方面创建更多的挑战。

15.1.2 规则

游戏规则用于规定玩家如何玩游戏，而且还暗示了对象的一些属性。用于Captain Blaster游戏的规则如下。

玩家将一直玩游戏，直到他们被流星击中为止。没有获胜条件。

玩家可以发射子弹来摧毁流星。每摧毁一颗流星，玩家将赢得1点。

玩家每秒可以发射两颗子弹。

玩家受屏幕各边限制。

流星将持续不断并且越来越快地出现，直到玩家输掉游戏为止。

15.1.3 需求

这款游戏的需求比较简单，如下所示。

作为外太空的背景纹理。
宇宙飞船模型和纹理。
流星模型和纹理。
游戏控制器，将在Unity中创建它。
有弹性的物理材质，将在Unity中创建它。
交互式脚本，将在MonoDevelop 中编写它们。

15.2 游戏世界

由于这款游戏是在太空中发生的，游戏世界实现起来相对比较简单。其思想是游戏将是2D的，并且贴图将在玩家背后垂直移动，看上去好像玩家在前进一样。实际上，玩家将是静止不动的。不过，在太空中翻滚之前，需要建立项目，可以从下面这些步骤开始。

（1）在名为Captain Blaster的文件夹中创建一个新项目，并向场景中添加一个定向灯光。

（2）创建一个Scenes文件夹，并把场景另存为Main。

（3）在Game视图中，把屏幕高宽比改为5:4，如图15.1所示。

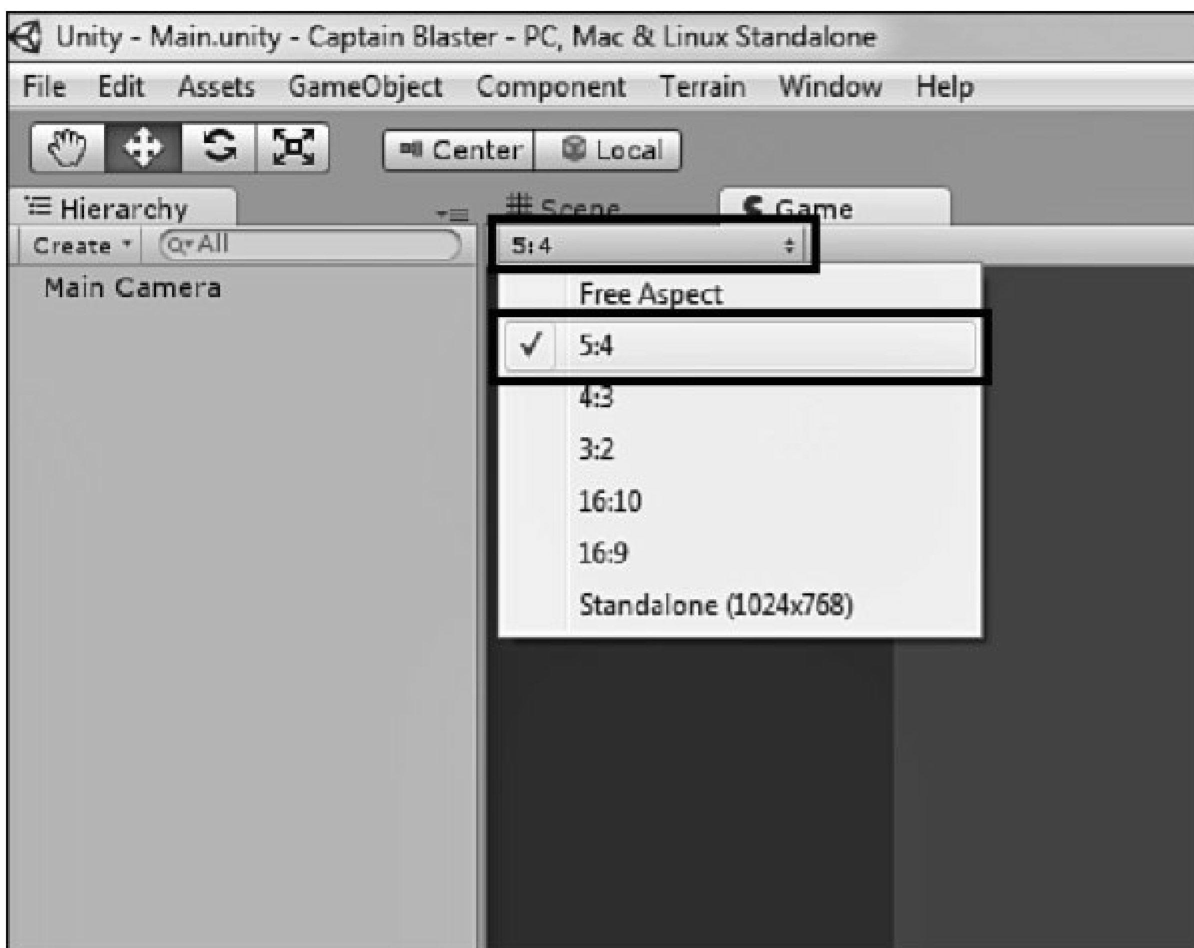


图15.1 设置游戏的屏幕高宽比

15.2.1 摄像机

既然已经正确地设置了场景，现在就应该处理摄像机了。在这里，你想要一部正交摄像机。这部摄像机缺少深度透视，非常适合于制作2D游戏。为了设置Main Camera，可以遵循下面这些步骤。

（1）把摄像机定位于(0, 0, −10)处，并且不进行旋转。

（2）把Projection属性改为Orthographic。

（3）把Size属性设置为6，如图15.2所示，查看摄像机的属性列表。



图15.2 Main Camera的属性

15.2.2 背景

正确设置滚动背景可能有点复杂。实质上，你将有两个背景对象在

屏幕上向下滚动。一旦底部的对象离开屏幕，就把它放在屏幕上方。在它们之间来回翻转，玩家将永远不会有所察觉。要创建滚动背景，可以遵循下面这些步骤。

（1）向场景中添加一个立方体，把它重命名为Background，并把它置于(0, 0, 0)处。然后把立方体缩放为(15, 15, .1)。

（2）在Project 视图中创建一个名为Textures 的新文件夹。在用于第15章（Hour 15）的本书配套资源中找到Star_Sky.png文件，并把它拖到新的Textures文件夹中。从Project视图中，把Star_Sky纹理拖到背景上。

（3）在Project 视图中创建一个名为Scripts的新文件夹。然后在该文件夹中创建一个名为BackgroundScript的新脚本，并把它拖到背景立方体上。在脚本中放入以下代码：

```
public float speed = -2;
// Use this for initialization
void Start (){
}
// Update is called once per frame
void Update (){
    transform.Translate(0f, speed * Time.deltaTime, 0f);
    if(transform.position.y <= -15)
    {
    transform.Translate(0f, 30f, 0f);
    }
}
```

（4）复制背景立方体，并把它放在(0, 15, 0)处。运行场景，你将注意到背景无缝地连续滚动出现。

注意：

无缝滚动

在前面的滚动背景中，你可能注意到一条短线。它出现的原因是用于背景的图像没有和场景拼贴在一起。一般来讲，它不是非常明显，并且游戏的动作将更多地把它掩盖住。不过，如果你在将来想要一种更无缝的背景，会希望使用一幅拼贴在一起的图像。

15.2.3 游戏实体

在这款游戏中，需要创建3个主要的实体：玩家、流星和子弹。这些项目之间的交互也非常简单。玩家发射子弹，子弹摧毁流星，流星摧毁玩家。由于从技术上讲，玩家可以发射大量的子弹，并且由于大量的流星可以再生，将需要一种方式清理它们。因此，还需要创建触发器，摧毁进入它们的子弹和流星。

15.2.4 玩家

你的玩家将是宇宙飞船。用于宇宙飞船和流星的模型已经由Duane Mayberry 提供给你了（http://www.duanesmind.co.uk），并且可以在用于第15章（Hour 15）的本书配套资源中找到它们。要创建玩家，可以遵循下面这些步骤。

（1）创建一个新文件夹，并把它命名为Meshes。在用于第15章（Hour 15）的本书配套资源中，找到名为Space Shooter的文件夹，并把它拖到新创建的Meshes文件夹中（以导入它）。

（2）在 Meshes 文件夹下，现在应该会有一个 Space Shooter 文件夹。找到其中的Space_Shooter.fbx 文件，并在编辑器中把比例因子改为0.09，如图 15.3 所示。一定要单击Inspector视图底部的Apply按钮。

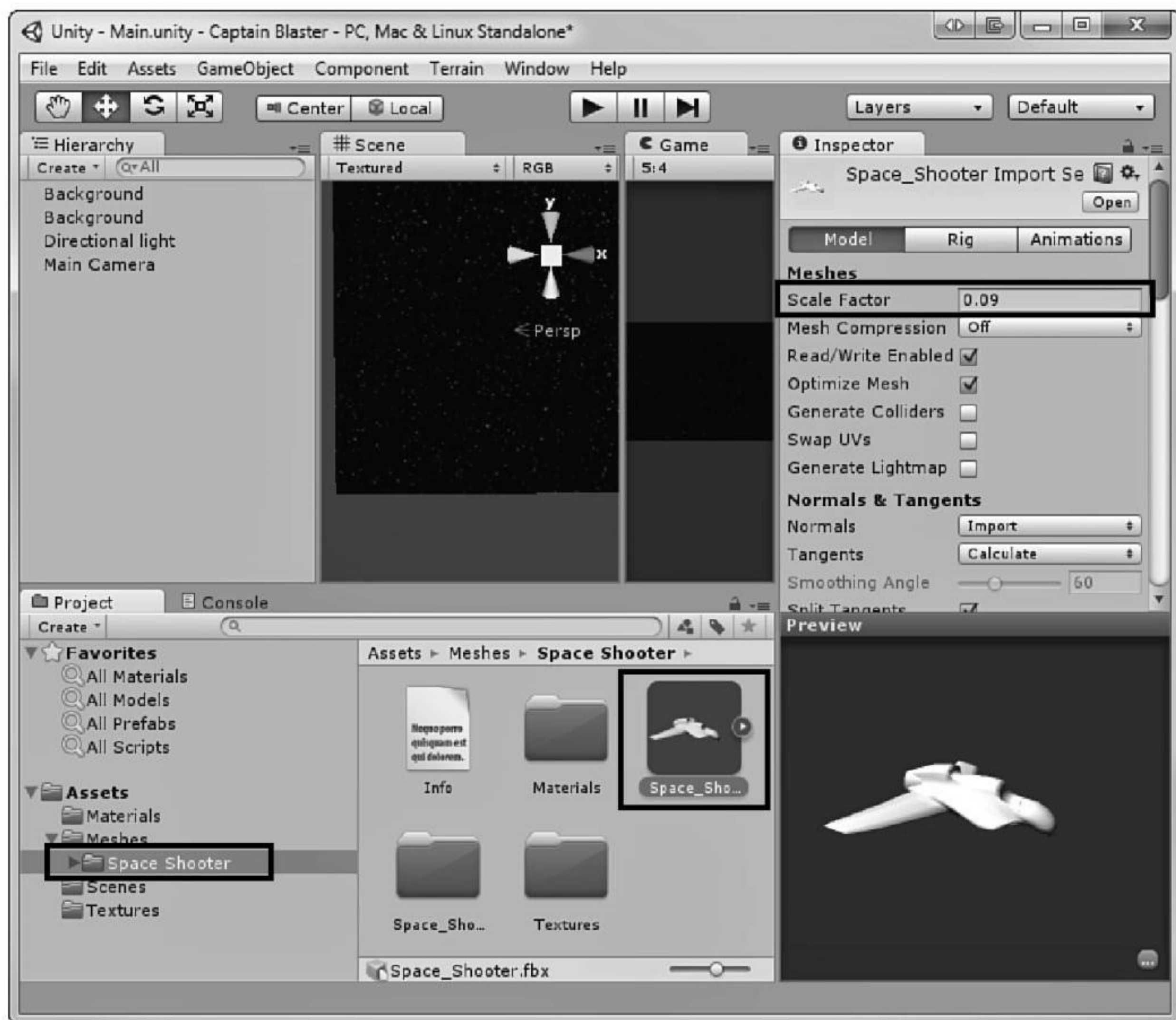


图15.3 太空射击模型

（3）从Project视图中单击Space_Shooter.fbx，并把它拖到Scene视图中。注意它面朝错误的方向。把它定位在(0, −4, −5)处，并把旋转角度设置为(270, 0, 0)。

（4）在Space Shooter文件夹下找到Textures 文件夹，然后单击1K_BodyTXTR.jpg 文件，并将其拖到Scene视图中的宇宙飞船模型上。

（5）给宇宙飞船添加一个胶囊碰撞器。选中Is Trigger 属性，然后把半径设置为0.62，把高度设置为1.71，并把方向设置为Z-Axis，如图15.4所示。

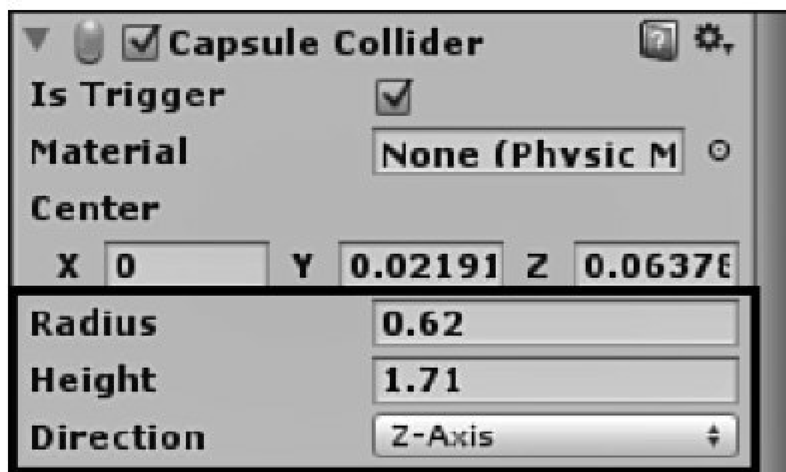


图15.4 胶囊碰撞器的设置

现在应该具有一只漂亮、纹理化并且面朝上的宇宙飞船，它准备摧毁一些流星！

15.2.5 流星

设置流星的步骤类似于设置宇宙飞船的那些步骤，唯一的区别是最后将把流星存放在一个预设中，以便以后使用。

（1）找到Meteor1文件夹，并把它拖到你在前面创建的Meshes文件夹中。

（2）在新的Meteor1文件夹中找到Meteor1.fbx文件，然后在Inspector视图中把比例因子改为0.5。一定要单击Inspector视图底部的Apply按钮。

（3）把Meteor1.fbx 文件拖到Scene 视图中。把它置于(0, 0, −5)处，然后把旋转角度设置为(0, 0, 0)，并把缩放比例设置为(1, 1, 1)（网格在导入时已经应用了某种旋转和缩放效果）。

（4）在Textures文件夹中，找到Meteor1_TXTR.png文件，并把它拖到场景中的流星上。

（5）给流星添加一个刚体，并且取消选中 Use Gravity 属性。给流

星也添加一个胶囊碰撞器。

（6）创建一个名为Prefabs的新文件夹，并在该文件夹中创建一个名为Meteor的新预设。从 Hierarchy 视图中单击 Meteor1 对象，并将其拖到新创建的预设上。然后从场景中删除Meteor1对象。

你现在就具有一个可重用的流星，它只等待造成严重的破坏。

15.2.6 子弹

在这款游戏中，子弹很简单。由于它们将非常快地移动，因此它们将不需要任何细节。要创建子弹，可以遵循下面这些步骤。

（1）向场景中添加一个胶囊，把它定位于(0, 0, 0)处，并把缩放比例设置为(.1, .1, .1)。给胶囊添加一个刚体，并取消选中Use Gravity属性。

（2）如果还没有Materials文件夹，就创建该文件夹，并在其中创建一种名为BulletMaterial的新材质。给该材质提供明亮的绿色，并把它应用于子弹。

（3）创建一个名为Bullet的新预设，然后单击胶囊并把它拖到子弹预设上。现在从场景中删除胶囊。

这是最后一个主要的实体。余下要创建的唯一一个实体是触发器，它们将阻止子弹和流星无休止地飞行。

15.2.7 触发器

触发器只是两个立方体，它们一上一下地放置在屏幕上，其职责就是捕获任何漂移的子弹和流星。

（1）向场景中添加一个立方体，并把它命名为Trigger。把该立方体定位在(0, −9, −5)处，并把缩放比例设置为(15, 1, 1)。

（2）在Inspector视图中，确保选中Box Collider 组件的Is Trigger 属

性。

（3）复制触发器，并把新的触发器放在(0, 9, −5)处。

现在所有的实体都已就位，可以开始把这个场景转变成游戏了。

15.3 控制

为了使这款游戏工作，将需要组合多个脚本组件。玩家需要移动飞船和发射子弹；子弹和流星需要能够自动移动；流星再生对象将使流星保持不断地流出；触发器将需要清理对象，还需要一个控制对象来跟踪所有的动作。

15.3.1 游戏控制

这款游戏中的游戏控制是基本的，因此要首先添加它。创建一个空的游戏对象，并把它命名为GameControl。创建一个名为GameControlScript的新脚本，并把它附加到游戏控制对象上。利用下面的代码覆盖脚本的代码：

```
using UnityEngine;
using System.Collections;
public class GameControlScript : MonoBehaviour {
  //is the game still going?
  bool isRunning = true;
  int playerScore = 0;
  void Start () {}
  void Update () {}
  public void AddScore()
  {
    playerScore++;
  }
```

```
    public void PlayerDied()
    {
        isRunning = false;
    }
    void OnGUI()
    {
        if(isRunning == true)
        {
            GUI.Label(new Rect(5, 5, 100, 30), "Player Score: " +
            playerScore);
        }
        else
        {
            GUI.Label(new Rect(Screen.width / 2 - 100, Screen.height / 2 -
            50, 200, 100), "Game Over. Your score was: " + playerScore);
        }
    }
}
```

在这段代码中，可以看到控制对象负责绘制GUI、保留分数以及知道何时游戏正在运行。控制对象具有两个公共函数：PlayerDied(　　)和AddScore()，其中前者是由玩家在流星击中它时调用的；后者则是由子弹在摧毁流星时调用的。最后，依赖于游戏状态绘制GUI。

15.3.2 流星脚本

流星实质上是从屏幕顶部往下落，并挡住玩家前进的道路。创建一个新脚本，并把它命名为MeteorScript。在Prefabs文件夹中，选择Meteor

预设。在Inspector视图中，找到Add Component按钮，如图15.5所示。然后单击Add Component > Scripts > Meteor Script 命令。



图15.5 Add Component 按钮

利用下面的代码覆盖流星脚本中的代码：

```
using UnityEngine;
using System.Collections;
public class MeteorScript : MonoBehaviour {
    float speed = -5f;
    //random rotation
    float rotation;
    void Start () {
        rotation = Random.Range(-40, 40);
    }
    void Update () {
        transform.Translate(0f, speed * Time.deltaTime, 0f);
        transform.Rotate(0f, rotation * Time.deltaTime, 0f);
    }
}
```

流星非常基本，它具有用于其速度和旋转的变量，旋转只用于使每颗流星相互之间看上去有点不同而已。在 Start()方法中，旋转被随机确定为一个−40～40 之间的数字。在Update()方法中，流星在屏幕上向下移动，并基于旋转变量围绕y轴旋转。注意，流星不负责确定碰撞。

15.3.3 流星再生

迄今为止，流星只是无法进入场景中的预设。你需要一个对象，负责以一定的时间间隔再生流星。创建一个新的空游戏对象，把该对象重命名为MeteorSpawn，并将其置于(0, 7, −5)处。然后创建一个名为MeteorSpawnScript的新脚本，并把它放在流星再生对象上。利用以下代

码覆盖该脚本中的代码：

```
using UnityEngine;
using System.Collections;
public class MeteorSpawnScript : MonoBehaviour {
    //meteor spawning timers
    float spawnThreshold = 100;
    float spawnDecrement = .1f;
    //meteor prefab
    public GameObject meteor;
    void Start () {}
    void Update () {
        //randomly determine if meteor spawns
        if(Random.Range(0, spawnThreshold) <= 1)
        {
            //create a meteor at a random x position
            Vector3 pos = transform.position;
            Instantiate(meteor, new Vector3(pos.x + Random.Range(-6, 6),
    pos.y, pos.z), Quaternion.identity);
            spawnThreshold -= spawnDecrement;
            if(spawnThreshold < 2)
            {
                spawnThreshold = 2;
            }
        }
    }
}
```

这个脚本正在做几件有趣的事情。第一件事是：它创建两个变量，

用于管理流星的时间选择。它还声明了一个 GameObject 变量，它将是流星预设。在 Update( )方法中，脚本生成一个随机数，该随机数在0与spawnThreshold变量（开始时是100）之间。如果随机数等于或小于1，就会再生流星。可以看到，再生流星的位置具有与再生点相同的y坐标和z坐标，但是x坐标会偏移一个数字，它在−6～6之间。这可以允许流星跨屏幕再生，而并非总是发生在相同的位置。最后，把spawnThreshold减少spawnDecrement。如果 spawnThreshold曾经到达2以下，就代之以把它设置为2。实际上，这段代码将随着时间的推移使流星再生得越来越快。由于总的范围在减小，随机获得一个1或以下的数字的可能性就会增加。因此，流星将更快地再生。

在Unity编辑器中，从Project视图中单击Meteor预设，并将其拖到流星再生对象的Meteor Spawn Script组件的Meteor 属性上。运行场景，你应该会注意到流星跨屏幕再生，它们刚开始时出现得比较慢。

15.3.4 触发器脚本

既然已经使流星可以在任意位置再生，开始清理它们就是一个好主意。创建一个名为TriggerScript 的新脚本，并把它同时附加到你在前面创建的上下两个触发器对象上。把以下代码添加到脚本中，确保代码位于方法之外，但是要位于类里面：

```
void OnTriggerEnter(Collider other)
{
   Destroy(other.gameObject);
}
```

这个基本的脚本只是简单地摧毁进入它的任何对象。由于玩家不能垂直移动，将不必担心它们被摧毁。只有子弹和流星可以进入触发器。

15.3.5 玩家脚本

目前，流星在下落，并且玩家无法避开它们。接下来需要创建一个脚本来控制玩家。创建一个名为PlayerScript的新脚本，并把它附加到宇宙飞船上。利用下面的代码替换脚本中的代码：

```
using UnityEngine;
using System.Collections;
public class PlayerScript : MonoBehaviour {
  //player speed
  public float speed = 10f;
  //bullet prefab
  public GameObject bullet;
  //Control Script
  public GameControlScript control;
  //player can fire a bullet every half second
  public float bulletThreshold = .5f;
  float elapsedTime = 0;
  void Start () {}
  void Update () {
    //keeping track of time for bullet firing
    elapsedTime += Time.deltaTime;
    //move the player sideways
    transform.Translate(Input.GetAxis("Horizontal") * speed *
    Time.deltaTime, 0f, 0f);
    //spacebar fires. The current setup calls this "Jump"
    //this was left to avoid confusion
    if(Input.GetButtonDown("Jump"))
```

```
        {
            //see if enough time has passed to fire a new bullet
            if(elapsedTime > bulletThreshold)
            {
                //fire bullet at current position
                //be sure the bullet is created in front of the player
                //so they don't collide
                Instantiate(bullet, new Vector3(transform.position.x,
                transform. position.y + 1.2f, -5f), Quaternion.identity);
                //reset bullet firing timer
                elapsedTime = 0f;
            }
        }
    }
    //if a meteor hits the player
    void OnTriggerEnter(Collider other)
    {
        Destroy(other.gameObject);
        control.PlayerDied();
        Destroy(this.gameObject);
    }
}
```

在这个脚本中做了许多工作。它首先创建了一些变量，分别用于速度、子弹预设、控制脚本和子弹的时间选择。

在Update( )方法中，脚本首先获取当前时间，它用于确定是否经过了足够长的时间，从而可以发射一颗子弹。如果你记得规则，就会知道玩家每隔半秒钟才能发射一颗子弹。然后，玩家将基于输入沿着 x 轴移

动。之后，脚本将确定玩家是否按下了空格键。通常，在 Unity中，空格键被视为跳跃动作。可以在Input Manager中指定它，但是这里保留了它的默认设置，以避免任何混淆。如果确定玩家按下了空格键，脚本就会检查相对于 bulletThreshold（目前是半秒钟）所流逝的时间。如果这个时间更长，脚本就会创建一颗子弹。注意：脚本只会在飞船上方一点创建子弹，这是为了防止子弹与飞船发生碰撞。最后，将流逝的时间重置为0，从而可以开启用于发射下一颗子弹的计数。

脚本的最后一部分包含 OnTriggerEnter( )方法。无论何时流星击中玩家，都会调用该方法。当发生这种情况时，流星将被摧毁，并且通知控制脚本玩家死亡了，然后摧毁玩家。

回到Unity编辑器中，单击并把子弹预设拖到玩家脚本的Bullet属性上。同样，单击并把游戏控制对象拖到玩家脚本上，从而允许它访问控制脚本，如图15.6所示。运行场景，并且注意你现在可以怎样移动玩家。玩家应该能够发射子弹（尽管它们不能移动）。另请注意：玩家现在可以死亡，并且结束游戏。

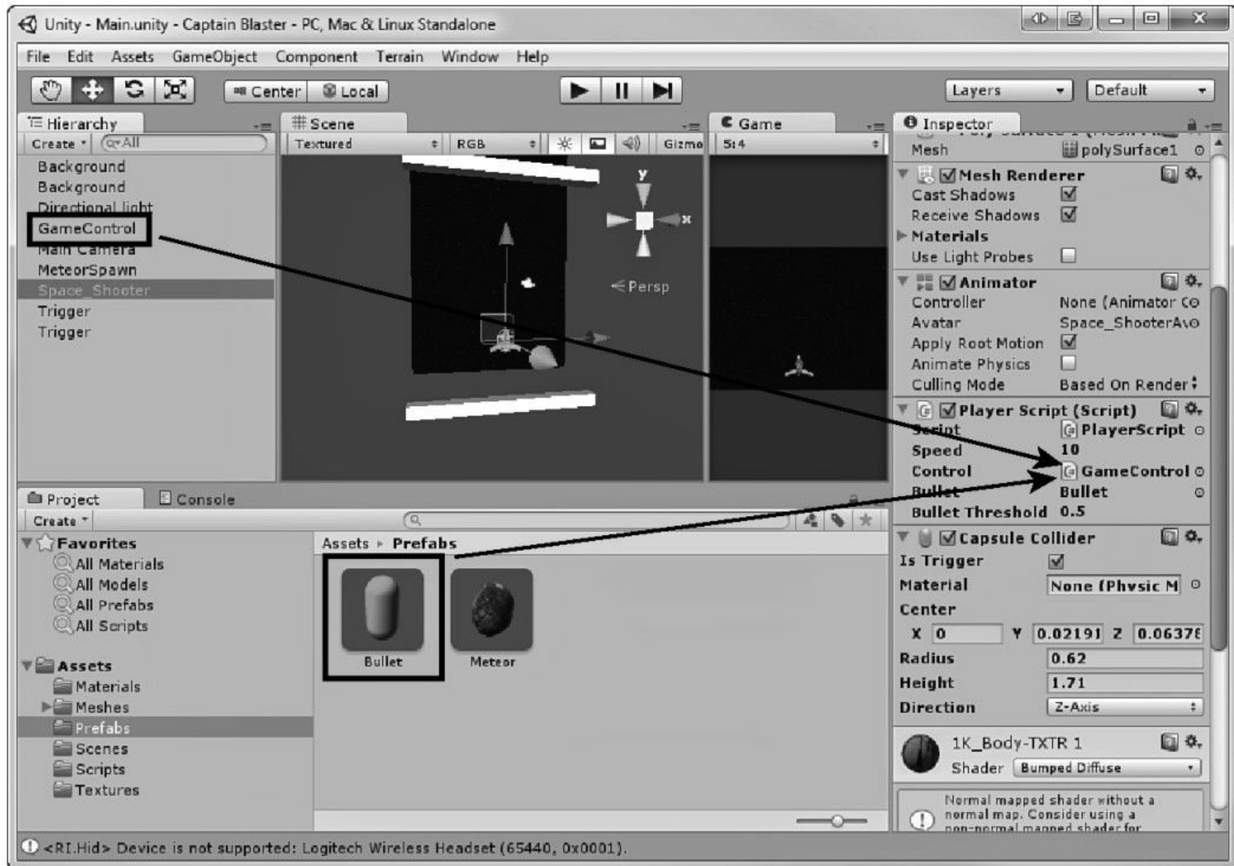


图15.6 连接玩家脚本

15.3.6 子弹脚本

你需要的最后一点交互性是使子弹移动并发生碰撞。创建一个名为BulletScript 的新脚本，并把它添加给子弹预设。然后利用下面的代码替换脚本中的代码：

```
using UnityEngine;
using System.Collections;
public class BulletScript : MonoBehaviour {
    float speed = 10f;
    //Game Control Script
    GameControlScript control;
```

```
    void Start () {
        //Because this is instantiated, we must find
        //the game control at run time
        control = GameObject.Find("GameControl").GetComponent<GameControlScript>();
    }
    void Update () {
        //move upward
        transform.Translate(0f, speed * Time.deltaTime, 0f);
    }
    //neither bullet nor meteor is a trigger, so we need
    //to use a different collision method here
    void OnCollisionEnter(Collision other)
    {
        Destroy(other.gameObject);
        control.AddScore();
        Destroy(this.gameObject);
    }
}
```

这个脚本与流星之间的主要区别是：这个脚本需要考虑碰撞和玩家得分。该脚本声明一个变量，用于保存控制脚本，就像玩家一样。不过，由于子弹实际上不在Scene视图中，它需要以一种稍微有点不同的方式访问控制脚本。在 Start()方法中，脚本将搜索 GameControl对象，然后调用GetComponent()方法，查找附加到其上的脚本。然后把控制脚本存储在control变量中。

由于子弹和流星上面都没有触发碰撞器，因此使用OnTriggerEnter()

方法将不会工作。作为替代，脚本使用了OnCollisionEnter( )方法。这个方法不会读入一个Collider变量，它将代之以读入一个Collision变量。在这种情况下，这两个方法之间的区别是不相关的。将要做的唯一工作是摧毁两个对象，并且把玩家的得分告诉给控制脚本。

继续前进并运行游戏，你将注意到游戏已经具备了可玩性。尽管你不能获胜（这是有意的），但是肯定可以输掉游戏。继续玩游戏，看看你最多能获得多高的分数！

15.4 改进

现在应该改进游戏。像以前的游戏一样，有意把多个位置保持为基本的。一定要从头至尾把游戏玩几次，看看你会注意到什么。哪些事情比较有趣？哪些事情没有趣味？有任何明显的方式中断游戏吗？注意：在游戏中留下了一个非常容易的骗招，允许玩家获得高分，你能找到它吗？

下面列出了一些你可以考虑改变的方面。

尝试修改子弹速度、发射子弹的延迟时间或者子弹的飞行路径。

尝试允许玩家并排发射两颗子弹。

尝试添加一种不同类型的流星。

给玩家提供额外的保护，甚至可能是防护罩。

允许玩家垂直以及水平移动。

这是一种常见的游戏题材，有许多种方式可以使之独具特色，尝试看一下你可以怎样自定义游戏。还值得指出的是，当你在本书后面学习粒子系统时，这款游戏将是用于试验它们的主要候选。

15.5 小结

在本章中，你制作了 Captain Blaster 游戏。你首先设计了游戏元素，接着构建了游戏世界。你构造了垂直滚动的背景，并使之运动起来。接下来，你构建了多个游戏实体，并通过编写脚本和控制对象添加了交互性。最后，你检查了游戏，并且探索了一些改进的方法。

15.6 问与答

问：流星应该这样缓慢地再生吗？

答：游戏将导致流星的再生速率随着时间的推移缓慢提高。如果它们对于你来说再生得太慢，可以自由地减小阈值。

问：Captain Blaster 游戏确实实现了船长的军衔吗，还是它只是一个名称？

答：很难讲，因为它基本上都是推断。有一件事情是确定的，它们不会把宇宙飞船提供给陆军中尉。

问：为什么把子弹发射的时间延迟半秒钟？

答：这主要是一个平衡的问题。如果玩家能够太快地射击，游戏将没有挑战性。

问：为什么在飞船上使用胶囊碰撞器？

答：高效、准确的碰撞检测可能比较困难。较大的碰撞器将覆盖机翼，有利于进行更准确的检测。不过，当流星出现在驾驶舱旁边时，这样的碰撞器将允许“错误的测试结果”。这样，它就是一种折衷。最佳的方式是使用多个碰撞器，以实现最大的准确性。应该避免使用这种方法，以使事情保持简单。

15.7 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

15.7.1 问题

1．游戏的获胜条件是什么？
2．滚动背景是怎样工作的？
3．哪些对象具有刚体？哪些对象具有碰撞器？
4．判断题：流星负责检测与玩家之间的碰撞。
5．玩家欺骗游戏的简单方式是什么？

15.7.2 答案

1．这是一个有意捉弄人的问题。玩家不能赢得游戏。不过，最高的分数将允许玩家在游戏之外“获胜”。

2．把两个具有相同纹理的立方体彼此堆叠在一起，然后它们将进行“蛙跳”穿过摄像机，看似无穷无尽。

3．子弹和流星具有刚体。子弹、流星、玩家和触发器具有碰撞器。

4．错误。

5．这仍然取决于你自己去查明它。这里只是提醒你，如果你还没有掌握这样的方式，可以自己去寻找它。

15.7.3 练习

与你迄今为止完成的练习相比，这个练习稍微奇怪一点。游戏改进

过程的一个常见部分是让没有参与开发过程的人测试游戏玩法，这允许完全不熟悉游戏的人给出诚实、第一次体验的反馈，这是极其有用的。这个练习是让其他人玩游戏。设法把各种各样的人组织在一起；设法找到一些热心的游戏玩家和一些不玩游戏的人；设法找到一些这种游戏体裁的粉丝和一些不喜欢这种体裁的人。把他们的反馈汇编成组，分为良好的特性、糟糕的特性以及可以改进的方面。此外，要设法查明是否有一些经常被请求的特性目前此游戏并不提供。最后，还要查明是否可以基于接收到的反馈来实现或改进游戏。

第16章 粒子系统

在本章中你将学到：

粒子系统的基本知识；

怎样处理模块；

怎样使用曲线编辑器。

在本章中，你将学习如何使用 Unity 的粒子系统，首先将从总体上了解粒子系统以及它们是如何工作的，其中将重点关注Unity新增的Shuriken粒子系统。接着，将试验许多不同的粒子系统模块。在本章最后，将试验Unity的曲线编辑器。

16.1 粒子系统

粒子系统实质上是一个对象或组件，它可以发射其他的对象，通常称为粒子（particle）。这些粒子有快有慢、有平面的也有具有某种形状的，并且有大有小。这个定义非常通用，因为这些系统利用正确的设置可以实现非常多的效果。它们可以创建喷射的火焰、滚滚的浓烟、萤火虫、雨、雾或者你可以想到的其他任何效果，这些效果通常称为粒子效果（particle effect）。

16.1.1 粒子

粒子是由粒子系统发射出的单个实体。由于许多粒子都是迅速发射出的，使粒子尽可能高效就很重要。这就是为什么大多数粒子都是 2D 广告牌的原因。记住，广告牌是一种总是面向摄像机的平面图像，这给它们提供了三维效果。

16.1.2 Unity粒子系统

从 update 3.5 起，Unity 就使用一种新的粒子引擎，称为 Shuriken 粒子系统。你创建的任何系统都将使用这个新的粒子引擎。不过，在 update 3.5 以前创建的粒子系统仍然会工作。在本章后面，你将实际地获得一个机会，试验作为Unity的一部分，提供一些遗留的粒子系统。

要在场景中创建一个粒子系统，可以添加一个粒子系统对象，或者给现有的对象添加一个粒子系统组件。要添加粒子系统对象，可以单击 GameObject > Create Other > Particle System命令。要给现有的对象添加粒子系统组件，可以选取该对象，并单击Component > Effects >Particle

System命令。

创建粒子系统

在这个练习中，将在场景中创建一个粒子系统对象。

（1）创建一个新的项目或场景。

（2）单击GameObject > Create Other > Particle System命令，添加一个粒子系统。

（3）在Scene视图中，注意粒子系统如何发射白色的粒子，如图16.1所示。这是基本的粒子系统。试着旋转和缩放粒子系统，看看它如何反应。

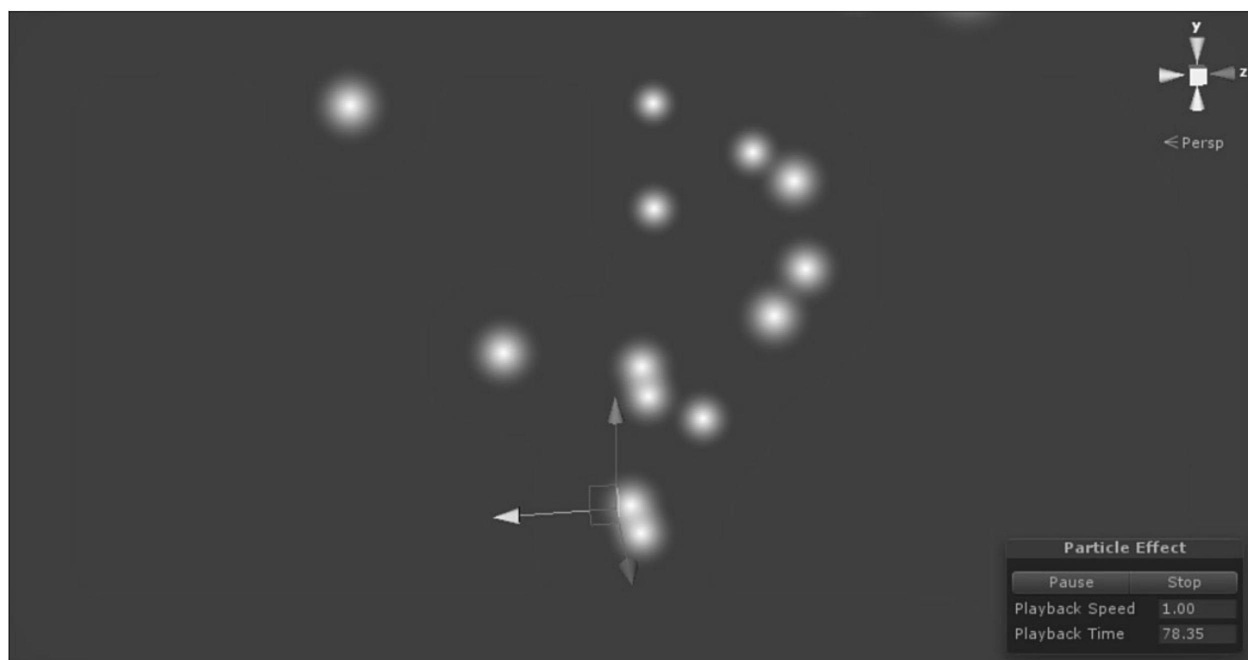


图16.1 基本的粒子系统

注意：

自定义的粒子

默认情况下，Unity中的粒子是小白色的球体，它们将逐渐消褪成透明的。这是一种确实有用的普通粒子，但它到目前为止只能如此。不过，你有时希望更具体一点（例如，创建火焰效果）。如果你愿意，可以通过任何2D图像创建你自己的粒子，以创建能够准确地适应要求的

效果。

16.1.3 粒子系统控制选项

你可能注意到，当向场景中添加粒子系统时，它将开始在Scene视图中发射粒子。你还可能注意到出现的粒子系统控制选项，如图16.2所示。这些控制选项允许在场景中暂停、停止和重新开始粒子动画，在调整粒子系统的行为组件时，这可能非常有帮助。

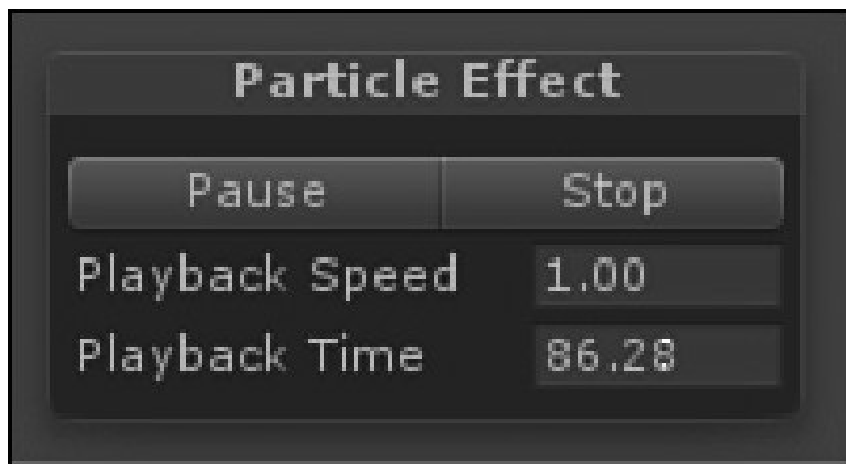


图16.2 粒子效果控制选项

控制选项还允许加快播放的速度，并且还会告诉你效果播放了多长时间。当测试持续的效果时，这可能证明非常有用。

注意：

粒子效果

要创建复杂的、引人注目的效果，你将希望多个粒子系统协同工作（例如，烟雾与火焰系统）。当多个粒子系统协同工作时，称之为粒子效果（particle effect）。在 Unity中，创建粒子效果是通过把粒子系统嵌套在一起实现的。一个粒子系统可以是另一个粒子系统的孩子，或者它们都可以是一个不同对象的孩子。Unity中的粒子效果最终都会被视作一个系统，并且粒子效果控制选项将把整个效果作为一个单元进行控制。

16.2 粒子系统模块

追根究底，粒子系统只是发射粒子对象的空间中的一个点。粒子的外观、行为以及它们产生的效果都是由模块确定的。模块是定义某种行为的多个属性。在 Unity 新增的 Shuriken系统中，模块是一个集成的、必要的组件。本节将列出每个模块，并且简要解释它用于做什么。注意：除了默认的模块（将最先介绍）之外，所有其他的模块都可以打开和关闭。要打开或关闭模块，可以选中模块的名称。要隐藏或显示模块，可以单击Particle System 模块旁边的加号（+），如图 16.3所示。默认情况下，所有的模块都是可见的，并且只启用了Emission、Shape和Renderer这些模块。要展开一个模块，只需单击其名称即可。

Inspector

Rotation

X 270 Y 0 Z 0

Scale

X 1 Y 1 Z 1

Particle System

Open Editor...

Particle System

Emission
Shape
Velocity over Lifetime
Limit Velocity over Lifetime
Force over Lifetime
Color over Lifetime
Color by Speed
Size over Lifetime
Size by Speed
Rotation over Lifetime
Rotation by Speed
External Forces
Collision
Sub Emitters
Texture Sheet Animation
Renderer

✓ Show All Modules

Particle System Curves

图16.3 显示所有的模块

注意：

简要说明

多个模块的一些属性要么是自解释的（比如矩形的长度和宽度属性），要么已经在前面介绍过。出于简单起见（以及为了防止本章的篇幅达到30页），将省略这些属性。如果在屏幕上看到比本书中所介绍的更多的属性，不要担心，这是有意为之的。

注意：

常量、曲线、随机

新的Shuriken系统引入了值曲线的概念。曲线允许在粒子系统的生存期内更改属性的值。通过值旁边的下指箭头，可以知道哪些属性能够使用曲线。所提供的选项是Constant、Curve、Random Between Two Constants和Random Between Two Curves。出于本节的目的，把所有的值都视作常量。在本章后面，你将有机会详细地探索曲线编辑器。

16.2.1 默认模块

默认模块被简单地标记为 Particle System。这个模块包含每个粒子系统都需要的所有特定的信息。表16.1描述了默认模块的属性。

表16.1 默认模块的属性

属性	描述
Duration	粒子系统将运行多长的时间（以秒为单位）
Looping	确定一旦持续时间到达粒子系统是否会重新开始运行
Prewarm	如果选择它，那么粒子系统在开始运行时，就好像它已经从前一个周期中发射了粒子一样
Start Delay	系统在发射粒子前将等待多长的时间（以秒为单位）
Start Lifetime	每个粒子将存活多长的时间（以秒为单位）
Start Speed	粒子的初始速度
Start Rotation	粒子的初始旋转角度
Start Color	所发射的粒子的颜色
Gravity Modifier	游戏世界给粒子应用了多大的重力
Inherit Velocity	在粒子上表现出了多快的系统速度（如果有的话）
Simulation Space	确定是在本地空间还是在游戏世界空间里模拟粒子
Play On Wake	确定是否在创建粒子系统时它就会立即开始发射粒子
Max Particles	一个系统每次可以存在的粒子总数。如果到达这个数字，系统就会停止发射，直到一些粒子死亡为止

16.2.2 Emission模块

Emission模块用于确定发射粒子的速率。使用这个模块，可以指定粒子是以恒定的速率或脉冲方式还是以它们二者之间的某种方式流出。表16.2描述了Emission模块的属性。

表16.2 Emission模块的属性

属性	描述
Rate	在一段时间或距离内发射的粒子数
Bursts	如果选择速率的时间选项，它将用于指定脉冲次数。通过单击加号（+）创建脉冲，并通过单击减号（–）删除脉冲，如图 16.4 所示

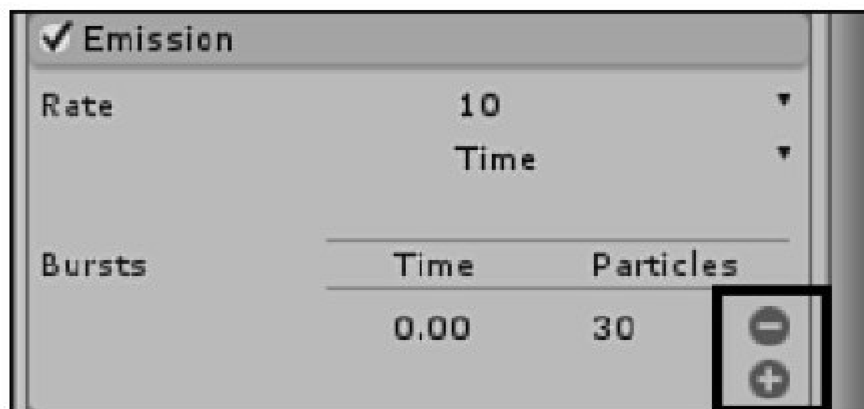


图16.4 Emission模块

16.2.3 Shape模块

顾名思义，Shape模块确定由发射的粒子构成的形状。形状选项有Sphere、Hemisphere、Cone、Box和Mesh。此外，每个形状还具有一组用于定义它的属性，这些属性比较普通，比如圆锥体和球体的半径。它们相当具有自解释性，这里将不会进行介绍。

16.2.4 Velocity over Lifetime模块

Velocity over Lifetime 模块通过对每个粒子应用x轴、y 轴和z轴速度，直接制作每个粒子的动画。注意：这是每个粒子在粒子（而不是粒子系统）的生存期内的速度变化。表16.3描述了Velocity over Lifetime 模块的属性。

表16.3 Velocity over Lifetime模块的属性

属性	描述
XYZ	应用于每个粒子的速度。它可以是一个常量、曲线，或者常量与曲线之间的一个随机数
Space	指定是基于本地空间还是游戏世界空间来添加速度

16.2.5 Limit Velocity over Lifetime模块

这个名称比较长的模块可用于抑制或固定粒子的速度。实质上，它将阻止粒子超过某根轴或所有轴上的速度阈值，或者降低它们的速度。表16.4描述了Limit Velocity over Lifetime模块的属性。

表16.4 Limit Velocity over Lifetime模块的属性

属性	描述
Separate Axis	如果不选中这个属性，那么它将为每根轴都使用相同的值。如果选中这个属性，则会显示每根轴的速度属性以及一个用于本地或世界空间的属性
Speed	每根轴或所有轴的速度阈值
Dampen	一个 0～1 之间的值，如果粒子的速度超过了由速度属性确定的阈值，就会把它的速度减慢这个值。值 0 根本不会减慢粒子的速度，但是值 1 将把粒子的速度减慢 100%

16.2.6 Force over Lifetime模块

Force over Lifetime 模块类似于Velocity over Lifetime 模块，其区别是：这个模块将给每个粒子应用一个力，而不是一个速度。该模块还允许相对于前面的所有帧，随机化每一帧上的力。

16.2.7 Color over Lifetime模块

Color over Lifetime 模块允许随着时间的流逝更改粒子的颜色。这可用于创建像火花这样的效果，它开始于明亮的橙色，并在消失前以暗红色结束。要使用这个模块，必须指定一种颜色渐变。也可以指定两种渐变，并使Unity在它们之间随机挑选一种颜色。可以使用Unity的渐变编辑器编辑渐变，如图16.5所示。

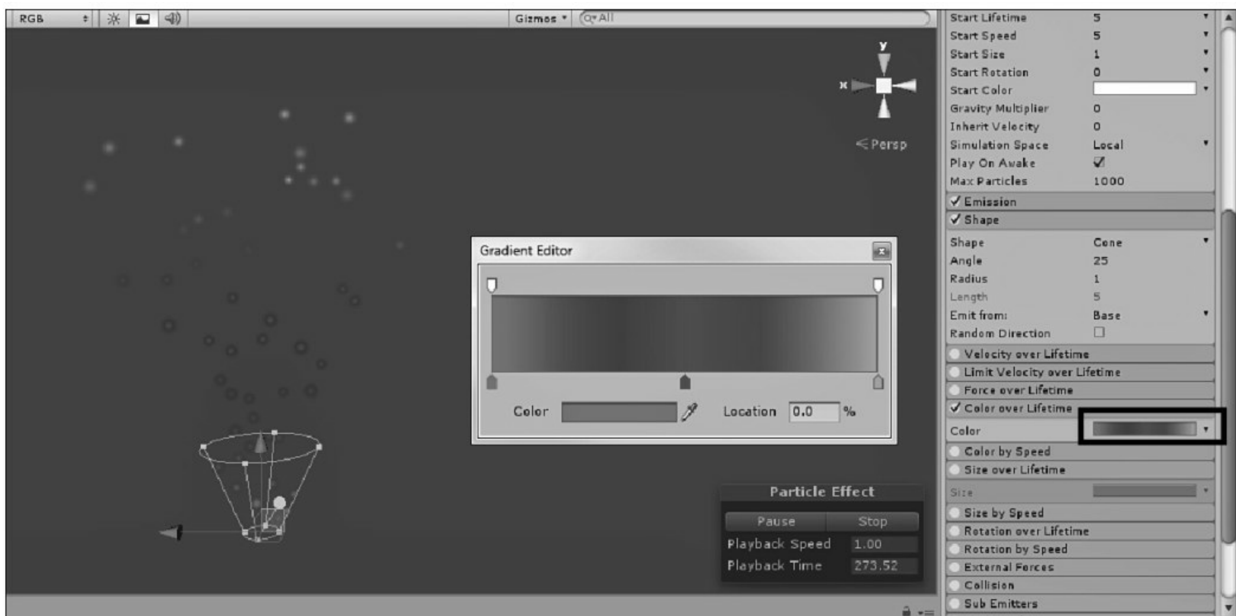


图16.5 渐变编辑器

注意：

渐变的颜色将与默认模块的Start Color 属性进行复合，这意味着如果起始颜色是黑色，那么这个模块将不起作用。

16.2.8 Color by Speed模块

Color by Speed 模块允许基于粒子的速度更改它的颜色。表16.5 描述了Color by Speed模块的属性。

表16.5 Color by Speed模块的属性

属性	描述
Color	用于指定粒子颜色的渐变（或者两种渐变，以便于随机选择颜色）
Speed Range	映射到颜色渐变的最低和最高速度值。以最低速度行进的粒子将映射到渐变的左边，具有（或超过）最高速度的粒子的颜色将映射到渐变的右边

16.2.9 Size over Lifetime模块

Size over Lifetime 模块允许指定粒子大小的变化。大小值必须是一条曲线，并且将指定随着时间的消逝粒子是将增大还是将收缩。

16.2.10 Size by Speed模块

与Color by Speed 模块非常像，Size by Speed 模块将基于粒子的速度（在最小值和最大值之间的速度值）更改它的大小。

16.2.11 Rotation over Lifetime模块

Rotation over Lifetime 模块允许指定粒子寿命内的旋转角度。注意：这个旋转是粒子本身的旋转，而不是世界坐标系统中的曲线的旋转。这意味着如果粒子是一个平面圆形，将不能看到旋转的效果。不过，如果粒子具有一些细节，将注意到它在旋转。用于旋转的值可以作为一个常量、曲线或者随机数给出。

16.2.12 Rotation by Speed 模块

Rotation by Speed 模块与Rotation Over Lifetime 模块相似，只不过它基于粒子的速度改变值。旋转将基于最小和最大速度值而改变。

16.2.13 External Forces模块

External Forces 模块允许对存在于粒子外部的任何力应用一个系数，它的一个良好示例是场景中可能存在的任何风力。Multiplier属性将依赖于力的值增大或减小力。

16.2.14 Collision模块

Collision模块允许为粒子设置碰撞。这可用于各类碰撞效果，比如火滚过墙壁或者雨击中地面。可以把碰撞设置为与预定的平面协同工作（Plane模式：最高效），或者与场景中的对象协同工作（World模式：

较慢的性能）。Collision模块具有一些公共属性和一些独特的属性，这依赖于所选的碰撞类型。表16.6描述了Collision模块的公共属性；表16.7和表16.8分别描述了属于Planes模式和World模式的属性。

表16.6 Collision模块的公共属性

属性	描述
Planes / World	指定使用的碰撞类型。Planes 模式将撞开预定的平面，World 模式将撞开场景中的任何对象
Dampen	确定当粒子发生碰撞时它的速度降低了多少，取值的范围是 0～1
Bounce	确定保留了多少速度的分量。与抑制不同，这只会影响粒子弹跳的轴。取值的范围是 0～1
Lifetime Loss	确定粒子在碰撞中失去了多少寿命。取值的范围是 0～1
Min Kill Speed	粒子在被碰撞销毁前的最低速度
Send Collision Messages	确定是否把碰撞消息发送给与粒子发生碰撞的对象

表16.7 Planes模式的属性

属性	描述
Planes	用于确定粒子可以碰撞什么位置的变换集合。所提供的变换的 y 轴确定了平面的旋转方式
Visualization	用于确定如果在 Scene 视图中绘制平面。它们可以是实心的，或者是网格
Scale Plane	调整平面的可视化区域的大小
Particle Radius	出于碰撞的目的，可用于使粒子看似更大或更小

表16.8 World模式的属性

属性	描述
Collides With	确定粒子碰撞的是哪些层，默认将其设置为 Everything
Collision Quality	游戏世界里的碰撞的质量，其值包括：High、Medium 和 Low。显然，High 是 CPU 密集的并且最准确，Low 则最次
Voxel Size	这个属性更高级，用于确定在媒介和低质量设置中使用的 Voxel 的密度。基本上把这个值保持为原样，除非你知道自己正在做什么

创建粒子碰撞效果

在这个练习中，将建立与粒子系统的碰撞。这个练习同时使用了Planes和World碰撞模式。

（1）创建一个新的项目或场景，向场景中添加一个粒子系统，并把它放置在(0, 0, 0)处。

（2）在Inspector中，单击Collision模块名称旁边的圆圈，启用这个

模块。单击Planes属性旁边的小加号（+），应该会出现一个平面，如图16.6　所示。你可能需要把视图改为Grid，以使之与图16.6中显示的内容匹配。注意粒子已经弹跳到平面上。四处移动和旋转平面，看看它如何影响粒子。

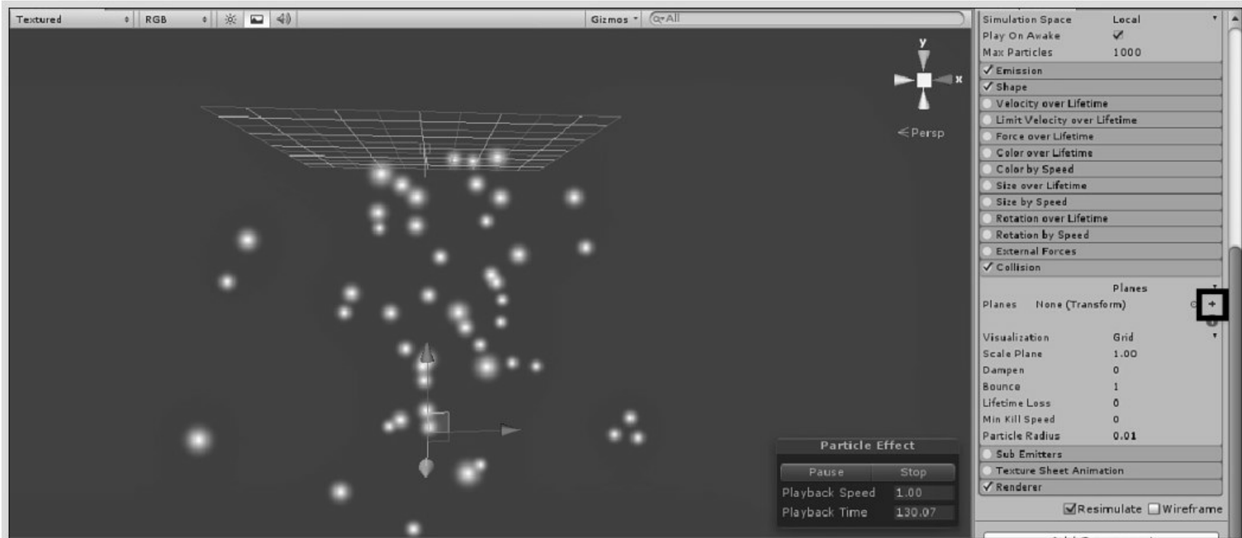


图16.6 添加平面变换

（3）向场景中添加一个立方体，把它定位于(0, 4, 0)处，并把缩放比例设置为(5, 1, 5)。

（4）注意粒子如何直接经过立方体。把Collision模块设置为World模式，如图16.7所示，注意粒子现在开始弹跳到立方体上。继续试验模块的不同属性，看看它们如何影响粒子。

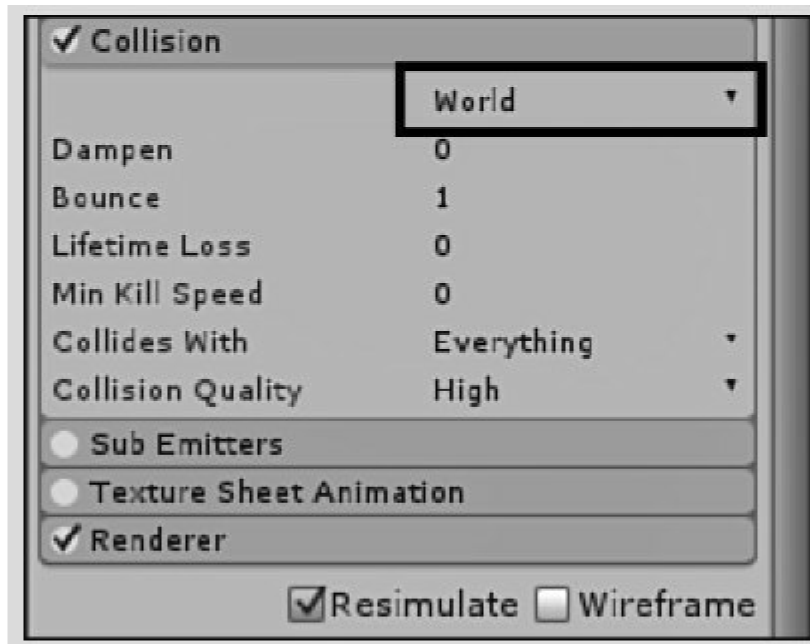


图16.7 把碰撞类型改为World模式

16.2.15 Sub Emitter模块

Sub Emitter 模块是一个极其强大的模块，使你能够在某些事件中为当前系统的每个粒子再生一个新的粒子系统。每当创建粒子或者粒子死亡或发生碰撞时，都可以创建一个新的粒子系统。这可用于生成复杂的、难以理解的效果（比如烟花）。这个模块具有3个属性：Birth、Death和 Collision，其中每个属性都将保存 0个或多个在各个事件上创建的粒子系统。

16.2.16 Texture Sheet 模块

Texture Sheet 模块允许更改在粒子的寿命中用于粒子的纹理坐标。实质上，这意味着可以把用于一个粒子的多种纹理放在单独一幅图像中，然后在粒子的寿命内在它们之间切换。表16.9 描述了Texture Sheet 模块的属性。

表16.9 Texture Sheet模块的属性

属性	描述
Tiles	确定纹理的拼贴方式
Animation	确定是整幅图像还是只有单独一排包含用于粒子的纹理
Cycles	指定动画的速度

16.2.17 Renderer 模块

Renderer模块指定实际上是怎样绘制纹理的，在这里可以指定用于粒子的纹理以及它们的其他绘图属性。表16.10描述了Renderer模块的属性。

表16.10 Renderer模块的属性

属性	描述
Render Mode	确定实际上是怎样绘制粒子的。模式有 Billboard、Stretched Billboard、Horizontal Billboard、Vertical Billboard 或 Mesh。所有的广告牌模式都会导致粒子与摄像机或者 3 根轴中的两根轴对齐。网格模式则导致在通过网格确定的 3D 中绘制粒子
Normal Direction	确定粒子在多大程度上面向摄像机。值 1 将导致粒子直接面向摄像机
Material	用于绘制粒子的材质
Sort Order	绘制粒子的顺序，可以是 None、By Distance、Youngest First 或 Oldest First
Sorting Fudge	确定绘制粒子系统的顺序。值越小，则越有可能在其他对象和粒子的顶部绘制粒子系统
Cast Shadows	确定粒子是否会投射阴影
Receive Shadows	确定粒子是否会接收阴影
Max Particle Size	设置最大的相对大小，值在 0～1 之间

16.3 曲线编辑器

前面列出的多个不同模块中的多个值都可以选择设置为 Constant 或 Curve。Constant 选项相当具有自解释性，给它提供一个值，它就是那个值。不过，如果希望那个值在经过一段时间后发生变化，则该怎么做？此时，使用新的曲线系统将很方便。使用这种特性，可以对一个值的表现方式进行非常精细的控制。在Inspector视图底部可以看到曲线编辑器，如图16.8所示。

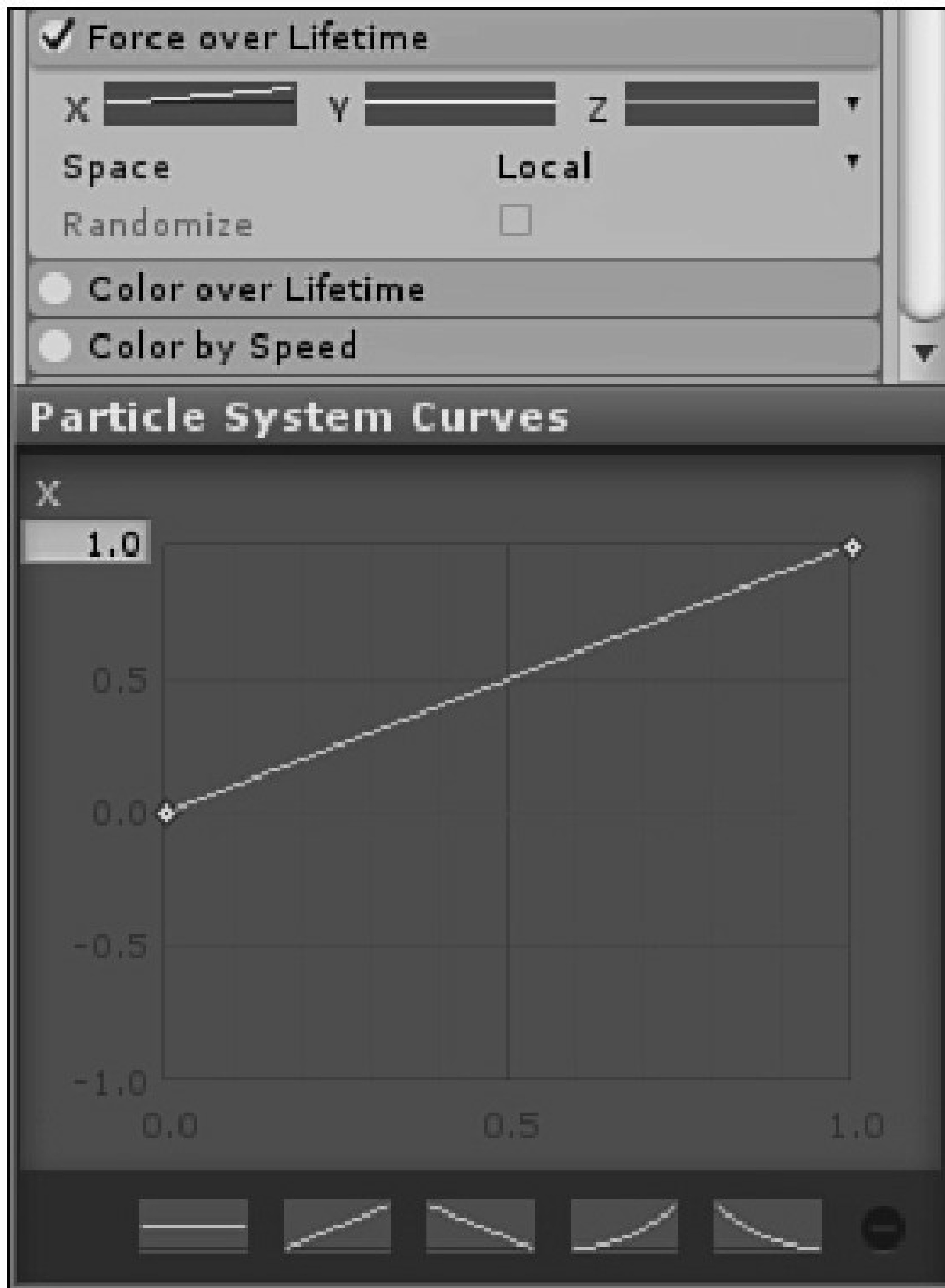


图16.8 曲线编辑器

曲线的标题是你正在确定的任何值。在图16.8中，值用于在Force over Lifetime 模块中沿着x轴应用的力。范围指定了可用的最小值和最

大值，可以修改它，以允许更大（或更小）的范围。曲线是在给定的一段时间内的值本身，预设则是可以提供给曲线的普通形状。

在任何关键点处都可以移动曲线，沿着曲线将这些关键点显示为可见的点。默认情况下，只有两个关键点：一个位于开头，一个位于结尾。可以右键单击曲线，并选择Add Key Point命令，在曲线上的任意位置添加新的关键点。

使用曲线编辑器

让我们熟悉一下曲线编辑器。在这个练习中，将更改在粒子系统的一个周期内发射的粒子的大小。

（1）创建一个新的项目或场景，然后添加一个粒子系统，并把它定位于(0, 0, 0)处。

（2）单击Start Size 属性旁边的下拉箭头，并选择Curve。

（3）把曲线的范围从1改为2。在中点附近右键单击曲线，并添加一个关键点。在曲线末尾做同样的事情。现在把中点拖到曲线编辑器的顶部，这将把值2赋予它，如图16.9所示。注意发射的粒子在粒子系统的5秒周期内如何改变大小。

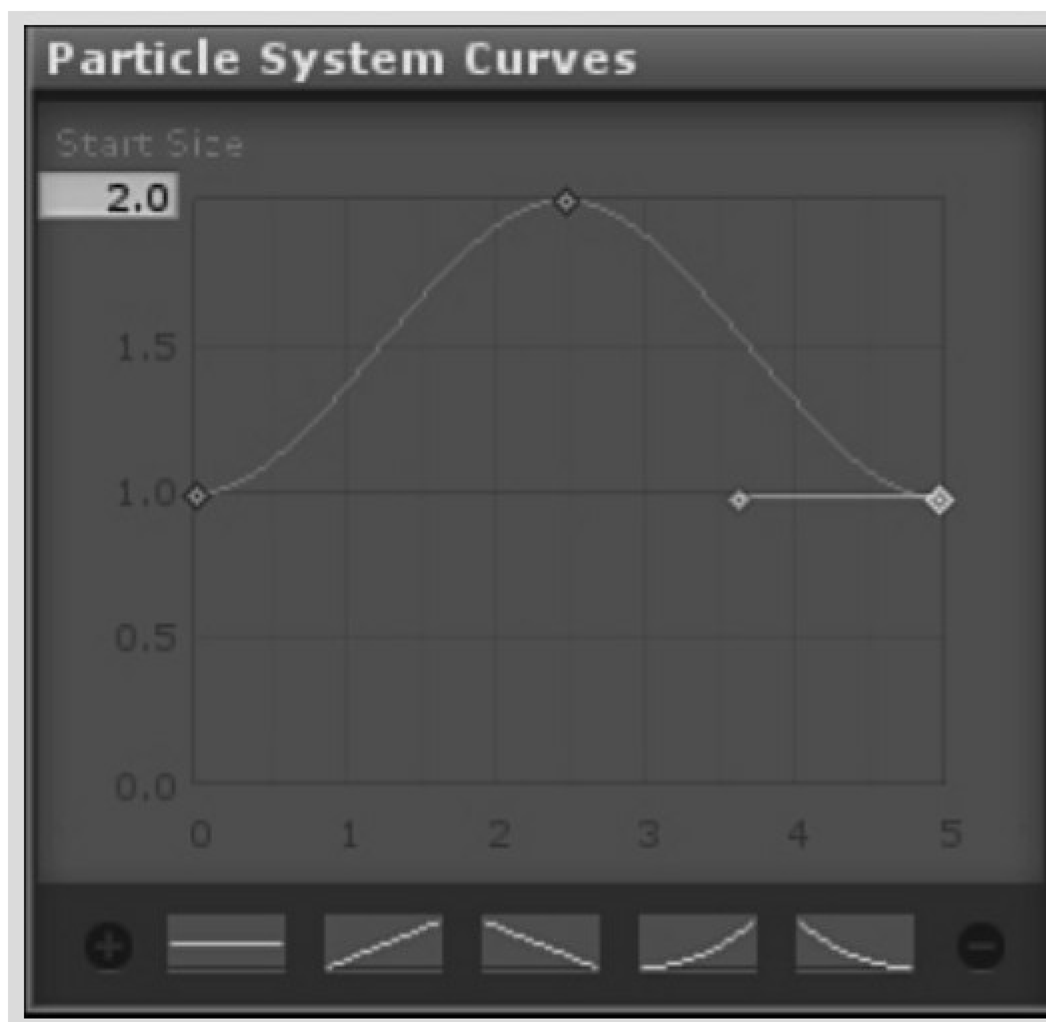


图16.9　Start Size曲线设置

16.4 小结

在本章中，向你介绍了Unity中的粒子系统。你学习了粒子和粒子系统的基础知识，然后详细检查了构成 Unity的粒子系统的许多模块。在本章最后还探讨了曲线编辑器的功能。

16.5 问与答

问：粒子系统是效率低下的吗？

答：它们可能是这样的，这依赖于你提供给它们的设置。一个很好的经验法则是：仅当粒子系统给你提供了某种价值时才使用它。它们可能看上去非常美妙，但是不要过度使用它。

16.6 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

16.6.1 问题

1．用于总是面向摄像机的2D图像的术语是什么？

2．Unity的新粒子系统的名称是什么？

3．哪个模块控制如何绘制粒子？

4．判断题：曲线编辑器用于创建会随着时间的推移而改变值的曲线。

16.6.2 答案

1．广告牌。

2．Shuriken粒子系统。

3．Renderer模块。

4．正确。

16.6.3 练习

在这个练习中，将试验一些现有的粒子效果，并尝试创建你自己的粒子效果。首先，要注意 Unity 包括的粒子效果是使用较老的系统创建的，这意味着它们将不具有相同的模块或设置。这非常好，可用于比较老系统相对于新系统是如何工作的。由于这个练习使你有机会同时处理现有的效果以及创建你自己的效果，因此没有正确的“解决方案”可供你查看，只需遵循这里的步骤并使用你的想象力即可。

1．单击Assets > Import Package > Particles命令，导入粒子效果程序包。一定要保持选中所有的资源，并单击Import按钮。

2．找到Fire 文件夹，它位于新建的Standard Assets 和Particles 文件夹下。单击Fire　和Flame预设，并把它们拖到场景中。试验这些效果的定位和设置。

3．继续试验所提供的其余的粒子效果（一定要检查Dust和Water效果）。

4．既然你已经看见了它可能是什么，现在就要看看你自己可以创建什么。试验多个不同的模块，并尝试提出你自己的自定义效果。

第17章 动画

在本章中你将学到：

动画的需求；

怎样为动画准备模型；

怎样应用动画；

怎样通过脚本触发动画。

在本章中，你将学习 Unity 中的动画。首先将准确了解动画是什么，以及需要什么才能使它们工作。之后，将探讨一种实际的模型，并且看看怎样使它为动画做好准备。接着，你将了解动画的细节，并把它应用于模型。在本章最后，将学习通过脚本触发动画。

注意：

动画系统

在 Unity中，有两个系统可用于动画。在本章中，将查看遗留的动画系统。你将精细地控制角色。不过，不要担心，在下一章中，将探讨新的Mecanim动画系统，并将使用新的、强大的Animator控制器。

17.1 动画的基本知识

动画是预制的可视运动集。在2D游戏中，这涉及具有多幅顺序的图像，并且可以非常快地翻转它们，结果就是对象看似在移动。这种效果类似于老式的翻转书。3D　　世界里的动画有很大的差别。在3D游戏中，使用模型来表示游戏实体。不能简单地在模型之间切换，以提供运动的幻觉。作为替代，将不得不实际地移动模型的某些部分，这将需要绑定（rig）和动画。

17.1.1 绑定

如果没有绑定，将不可能把模型制作成动画（或者困难得超乎想象）。其原因是：如果没有绑定，计算机将无法知道模型的哪些部分被指望移动，以及指望它们如何移动。那么，准确地讲，绑定是什么？与人体骨骼非常相似，如图17.1所示，绑定指定了模型的哪些部分是刚性的，它们通常称为骨头（bone）。它还指定了哪些部分可以弯曲，这些可弯曲的部分称为关节（joint）。

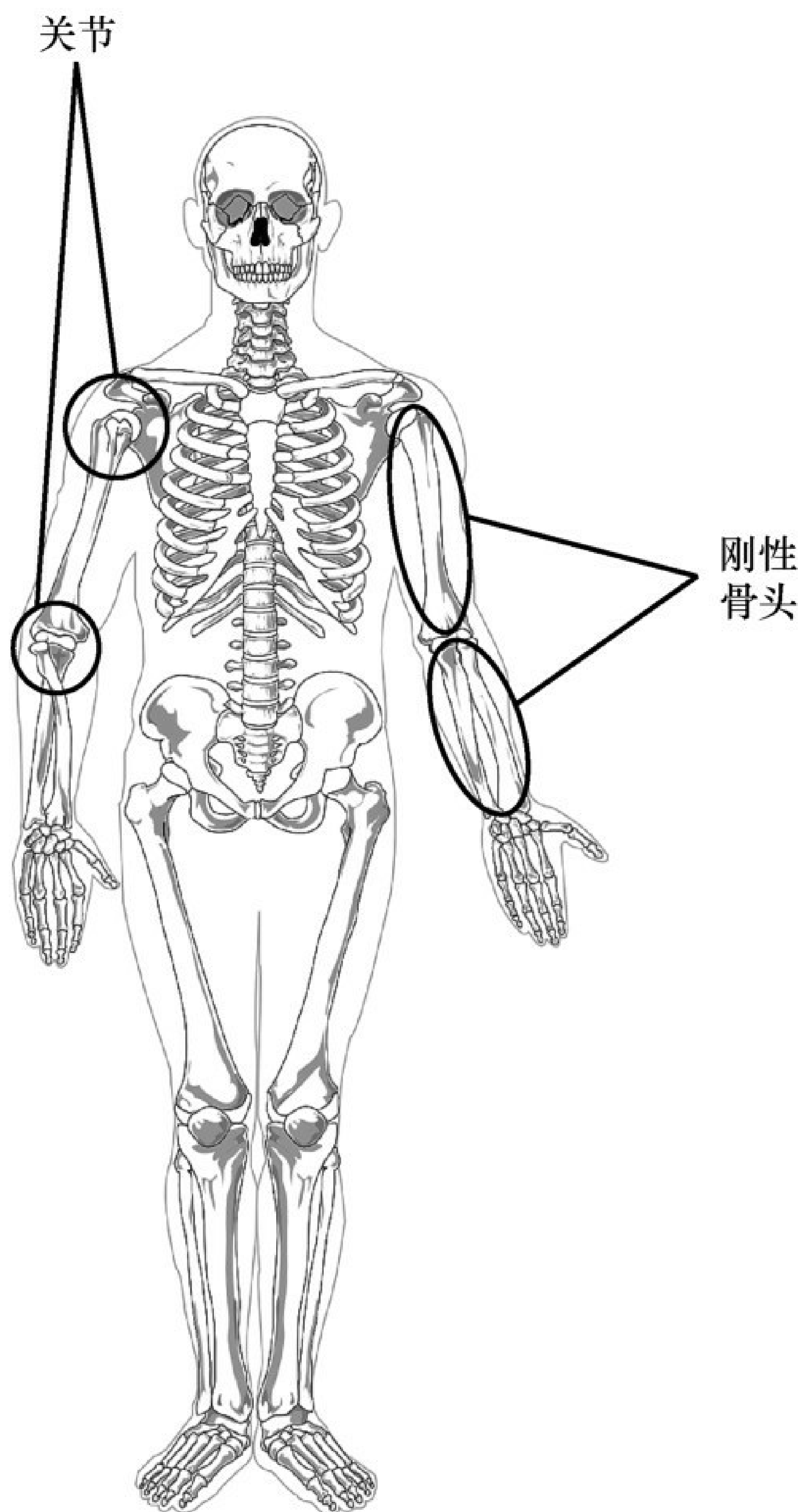

关节
刚性
骨头

图17.1 骨骼作为一种绑定

骨头和关节协同工作，定义了模型的物理结构。将使用这种结构实际地定义模型的动画。

17.1.2 动画

一旦模型具有绑定，就可以给它提供一个动画。在技术层面上，动画只是一系列用于绑定的指令（放入右手、拿出右手、现在放入右手并摇晃它……）。这些指令可以像电影一样播放，可以暂停、单步调试或逆转它们。此外，利用正确的绑定，更改模型的动作就像更改动画一样简单。有时，这些动画可以是带有指令的事件，用于在 3D 空间里移动整个模型。最好的部分是：如果具有两个完全不同的模型，但是它们具有相同的绑定，那么就可以用完全相同的方式对它们应用相同的动画。因此，半兽人、人类、巨人和狼人都可以执行完全相同的舞蹈动画。

注意：

想要的3D艺术师

关于动画的事实是：大部分工作是在像Unity这样的程序外面完成的。一般来讲，建模、纹理化、绑定（rigging）和动画都是由称为 3D 艺术师的专业人员在诸如Blender、Maya、3D Studio Max之类的程序或者任何其他的 3D 创作软件中创建的。因此，在本书中没有介绍它们的创建。作为替代，本书将说明如何获取已经创建的资源，并在 Unity中把它们结合起来，构建交互式体验。记住：制作游戏不仅仅是把各个部分组合起来。你可能会使游戏工作，但是艺术师可以使它变得好看！

17.2 为动画准备模型

正确地为动画准备模型不需要做太多的工作。如果顺利的话，绑定模型的过程应该已经完成了。如果还没有，将需要在导入 Unity 之前完成它。在本节中，你将开始处理已经绑定的模型，并将获得专门为模型制作的动画。在真实的生产环境中，你或其他一些 3D 艺术师将不得不开发这些项目，之后才能在Unity中使用它们。

在本节中，将从Unity Asset Store 中获得一个模型。这个模型带有许多不同的项目，你将仔细检查每个部分，以确保它们配置正确。要访问Unity Asset Store，可以单击Window >Asset Store命令。可能会要求你登录。如果是这样，就可以使用你在第1章中创建的登录账号。一旦加载了Asset Store，就可以找到搜索窗口，并搜索“Warrior”，如图 17.2 所示。

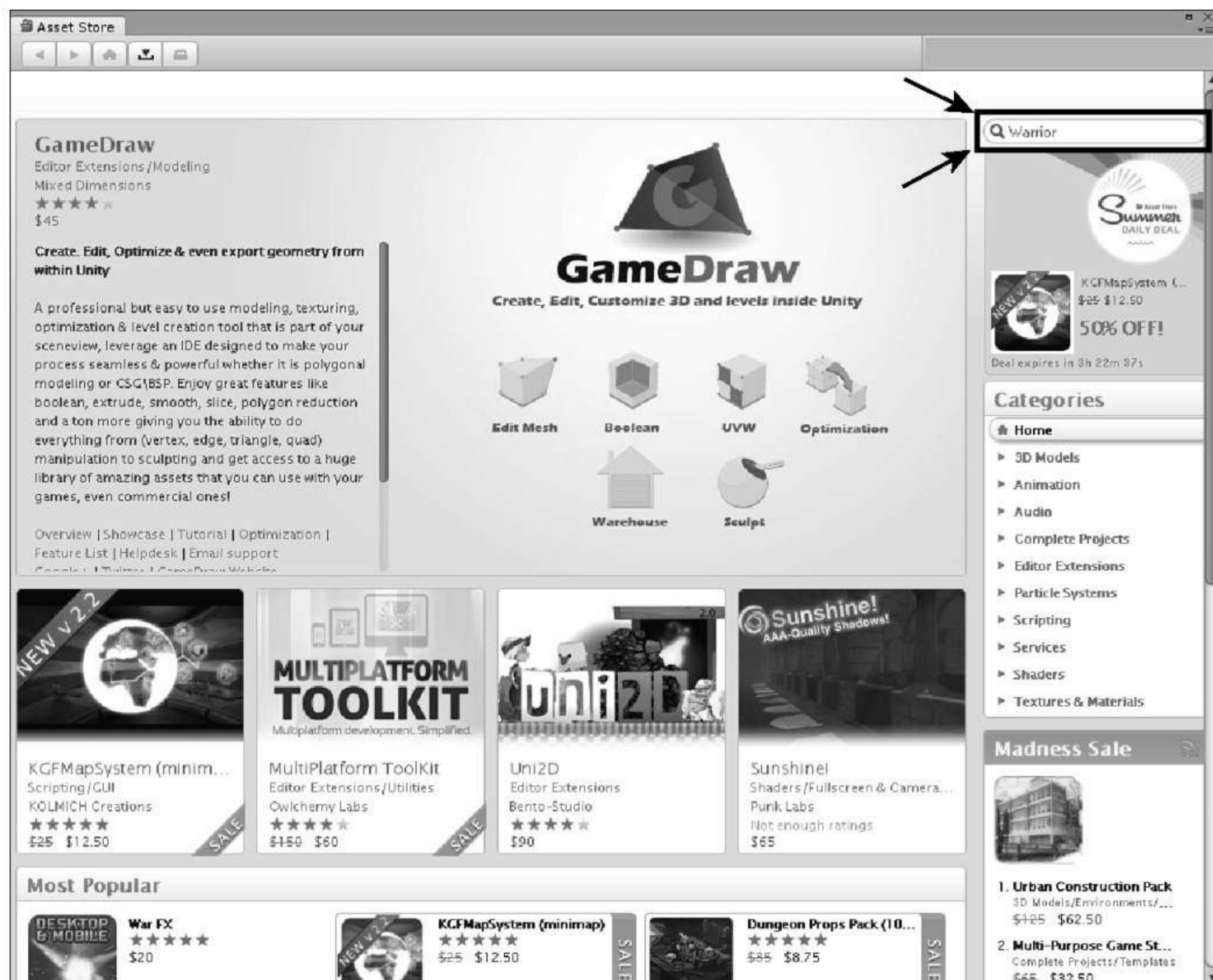


图17.2 Asset Store 的搜索栏

找到免费的模型3dsmax Bip Warrior Anim Free，并单击Import，如图 17.3 所示。当Import Package对话框出现时，确保选择所有的选项，并单击Import按钮。



图17.3 需要的模型

你现在应该在Project 视图中注意到一个名为“3dsmax Bip Warrior Anim Free”的新文件夹。花一点时间，让自己熟悉一下该文件夹的内容，在本章余下部分将使用它们。

注意：

演示

士兵模型带有一个演示场景，在3dsmax Bip Warrior Anim Free文件夹下的Scenes文件夹中可以找到它。打开这个场景，将允许应用多种动画来测试模型。

17.2.1 模型

在新创建的3dsmax Bip Warrior Anim Free 文件夹下的Models 文件夹中，可以找到你将使用的模型，它被命名为Soldier_f_0。找到该模型，并选取它。在Inspector视图中，应该会看到3个主要的选项卡：Model、Rig和Animations，如图17.4所示。

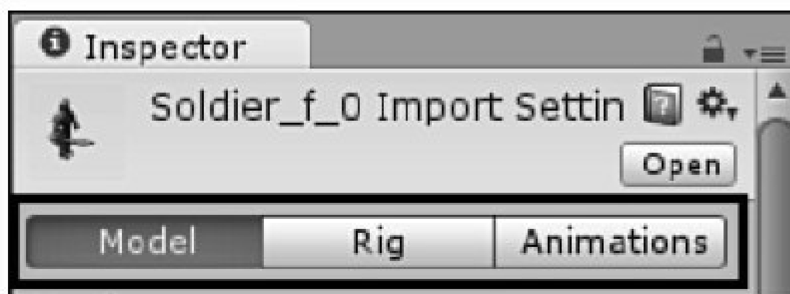


图17.4 模型的 Inspector视图

Model 选项卡负责所有的设置，它们指定了如何把模型自身导入到Unity 中。出于本章的目的，可以安全地忽略这些项目。你将关注的两个选项卡是Rig和Animations选项卡。在Rig 选项卡下，确保将Animation Type属性设置为Legacy，并将 Generation 属性设置为Store in Root (New)。图17.5显示了正确的设置。

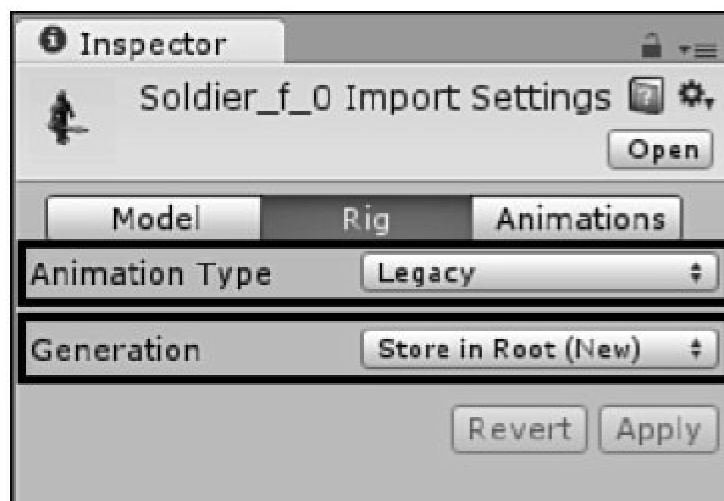


图17.5 绑定设置

接下来是Animations选项卡。动画通常是作为模型的一部分出现的，这很好，因为你不必管理多个文件，并且可以代之以把所有内容都打包在一起。Animations 选项卡包含管理内置的动画所需的所有属性和

控制选项。不过，出于本书的目的以及为了方便学习，将需要禁用它们。在Animations选项卡下，取消选中Import Animation 复选框，然后单击Apply按钮。你的屏幕现在应该为如图17.6所示的样子。

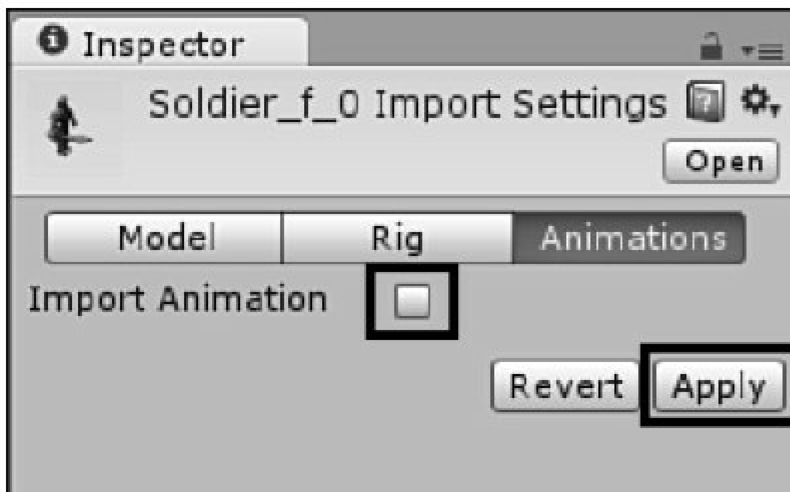


图17.6 动画设置

提示：

模型被绑定了吗？

你可能感到迷惑的是，怎样知道一个模型是否被绑定。最容易的方式是询问任何制作模型的人。如果这不可能，那么总是可以在Hierarchy 视图中查看模型。一般来讲，绑定的模型将包含多个子游戏对象，这些对象将与多个绑定关节对应。图17.7显示了一些子对象，可以使用它们确定模型是否被绑定。

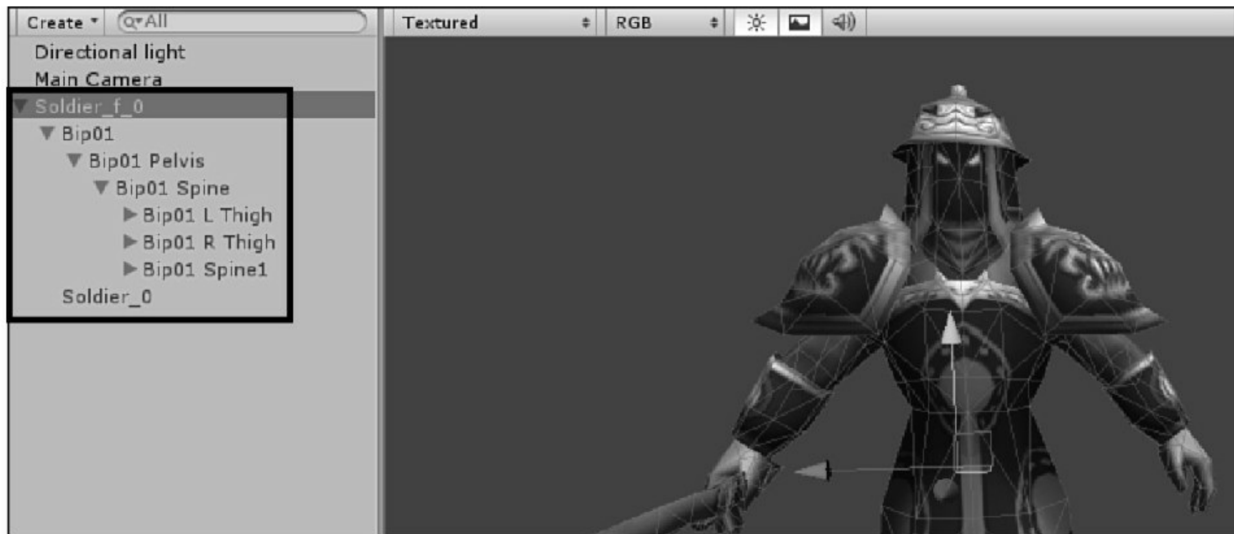


图17.7 士兵子对象

17.2.2 动画资源

你想做的下一件事情是仔细检查动画资源，并且确保按你想要的方式设置它们。在3dsmax Bip Warrior Anim Free 文件夹下找到Animations文件夹，该文件夹包含4 个可用的动画：death、idle0、idle1 和idle2。选取其中的任何动画资源，将允许修改它们的Wrap Mode属性并预览它们。不过，在预览动画之前，需要提供一个模型。图17.8显示了在提供模型之前的Preview窗口的外观。



图17.8 没有提供可用模型的Preview窗口

为了修正它，只需从Models文件夹中单击Soldier_f_0模型，并把它拖到动画的Preview窗口中即可，如图17.9所示。

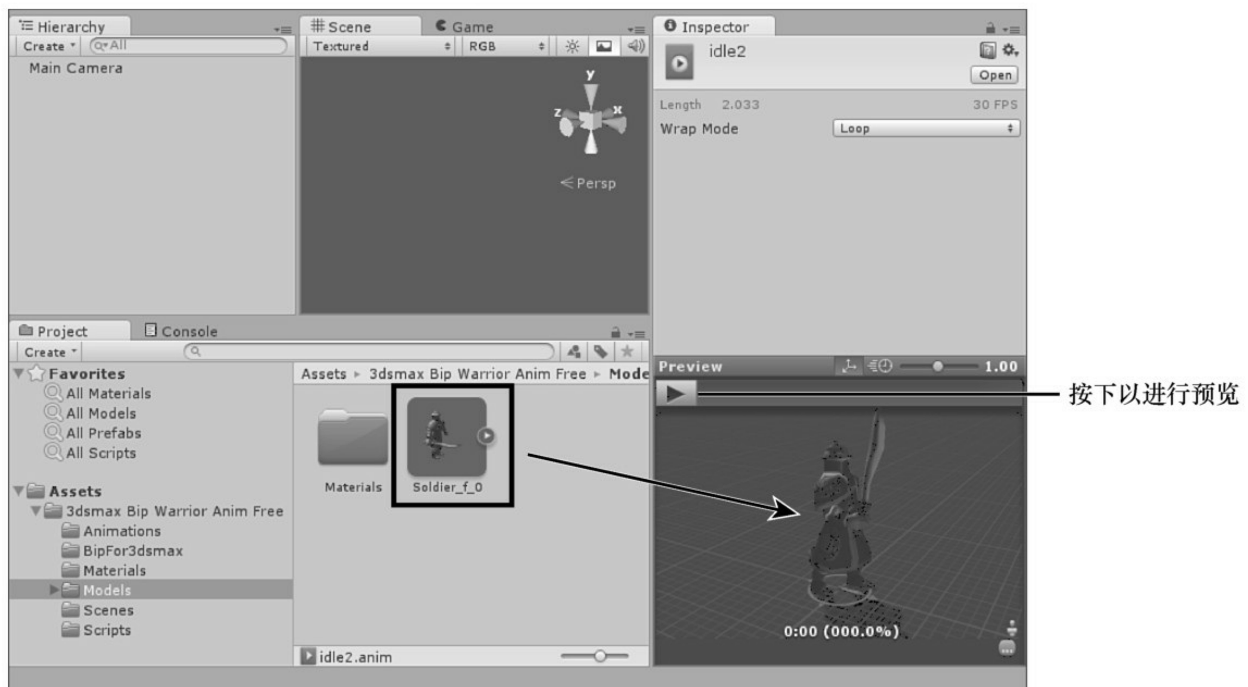


图17.9 给预览窗口添加一个模型

一旦完成该操作，只需单击Play按钮，即可预览那个模型上的动

画。花一点时间，预览全部4 个动画。目前不要担心Wrap Mode 属性，在本章后面将更详细地介绍它。

把士兵添加到场景中

让我们把士兵模型添加到场景中并试验它。在这个练习中创建的场景将会在后面使用，因此一定要保存它。

（1）创建一个新的项目或场景，并向场景中添加一个定向灯光。

（2）找到Soldier_f_0 模型资源，并把它拖到Scene 视图中。把新建的士兵定位于(0, 0, −5)处，并把旋转角度设置为(0, 180, 0)。确保场景中的士兵在 Inspector 视图中具有一个Animation组件，任何其他的组件（如Animator）都意味着上一步完成得不正确。

（3）运行场景，注意士兵模型就在那里，但是没有移动。这正是你想要的，因为还没有实际地应用动画。一定要保存这个场景，以便以后使用。

17.3 应用动画

如你在上一节中所看到的，士兵模型附加了一个 Animation 组件。这个组件就像一组不同动画的集合，可以在运行时应用到模型上。表17.1描述了Animation组件的属性。

表17.1 Animation组件的属性

属性	描述
Animation	这是用于模型的默认动画。如果没有在这里指定一个动画，那么模型将不会是动画式的
Animations	这是可以应用于这个模型的所有动画的列表，可以通过把动画拖到这个属性上来添加它们
Play Automatically	如果启用这个属性，将导致模型在创建时开始运行默认的动画
Animate Physics	这个属性确定动画是否会与物理学交互
Culling Type	这个属性确定动画何时开始播放，选项有 Always Animate、Based on Renderer、Based on Clip Bounds 和 Based on User Bounds。一般来讲，可以把它保留为默认设置，除非你尝试实现特定的功能

17.3.1 添加动画

如前所述，可以通过把动画拖到Animation组件的Animations属性上，把它们应用于模型，如图17.10所示。

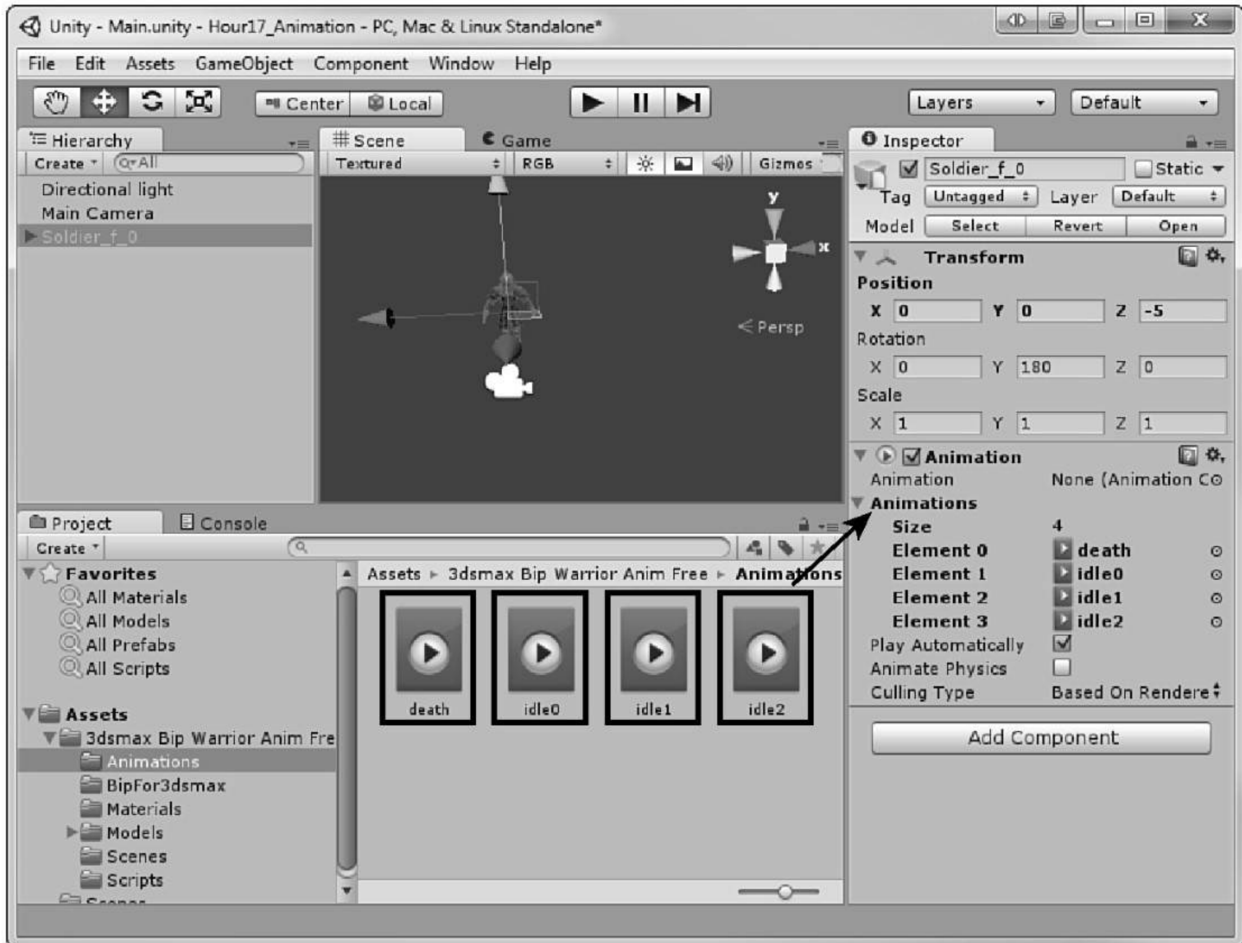


图17.10 把动画添加给模型

这样做只是使动画可供模型使用。然而，它不会实际地创建模型的动画。在多个动画之间切换是通过脚本完成的，这将在下一节中介绍。目前，你可能非常渴望实际地看看一个动画是如何工作的。要查看动画在模型上运行，可以把想要的动画拖到 Animation 组件的Animation 属性上。这将为模型设置默认的动画。确保也设置Play Automatically选项。现在，当运行场景时，将会看到模型在实际地移动！

制作模型的动画

这个练习使用你在前面创建的场景。如果还没有创建该场景，就要回过头去完成前面的练习“把士兵添加到场景中”。在下面这个练习中，将创建士兵的动画。完成后，一定要保存这个场景，因为将在后面的练习中使用它。

（1）打开以前创建的场景。

（2）把每个动画都拖到 Animation 组件的 Animations 属性上，并把 Idle0 动画拖到Animation组件的Animation属性上。

（3）播放场景，注意 Idle0 动画怎样开始播放士兵。停止场景，并返回到编辑器中，试验模型上的每个动画，看看它们都是如何运行的。另请注意 death 动画不会循环播放，而是在完成后停止播放。

17.3.2 包装模式

在本章前面，你查看了Inspector视图中的动画资源，可能注意到有一个你忽略了的名为Wrap Mode的属性。实质上讲，包装模式确定了一旦动画完成运行它将会做什么。表17.2 描述了5种不同的模式。

表17.2 包装模式

属性	描述
Default	这个选项允许动画做任何默认的事情，通常默认为只播放一次
Once	动画播放一次，然后停止
Loop	动画在到达末尾时重新开始播放
Ping Pong	动画在到达末尾时再次运行。不过，其区别是动画向后运行。动画将继续向前循环，然后向后运行，周而复始
Clamp Forever	动画运行一次。当动画到达末尾时，它将继续反复地播放最后一帧。这有别于 Once 属性，因为后者只会播放一次，然后回到第一帧

使用包装模式

这个练习使用你在前面创建的场景。如果还没有创建该场景，就要回过头去完成前面的练习“制作模型的动画”。在下面这个练习中，将处理不同的包装模式。

（1）打开以前创建的场景。

（2）确保 Idle0 动画被设置为士兵模型的默认动画，然后播放场景，并且注意士兵如何移动。

（3）在Project 视图中，找到Idle0 动画资源，并把Wrap Mode属性改为Ping Pong。再次运行场景，并且注意模型如何不同地移动。

（4）继续试验不同的动画和包装模式。

17.4 编写动画的脚本

尽管使模型运行一个循环的空闲动画比较不错，但它可能有点令人厌烦。更可能的是，你需要模型不仅仅只是循环运行相同的动画。如你以前所看到的，可以给 Animation 组件的Animations属性添加一串动画。不过，你没有看到的是，在场景运行时如何在这些动画之间切换。通过编写脚本可以实现许多复杂的功能，但是在本节中重点关注的将只是播放动画。下一章将介绍更高级的动画脚本编程。

要在模型上播放动画，需要使用transform.animation.Play( )方法。例如，要播放一个名为walk的动画，可以编写以下代码：

```
transform.animation.Play("walk");
```

通过简单地播放另一个动画，可以随时更改动画。如果一个动画被设置为循环，它将自动这样做。

更改动画

在这个练习中，当场景在运行时，通过脚本更改模型动画。你将在Start( )方法中更改这个模型的动画。这个练习将使用在前面的练习“向场景中添加士兵”中创建的场景。

（1）打开以前创建的场景，确保 idle0 是士兵的默认动画，并且确保利用你一直在使用的4个动画填充Animation组件的Animations属性。

（2）创建一个名为 SoldierScript 的新脚本，并把它附加到当前位于场景中的士兵模型上。然后向脚本的Start()方法中添加以下代码：

```
void Start(){
  transform.animation.Play("idle1");
}
```

（3）运行场景，并且注意角色在循环播放 idle1 动画。停止场景，

并返回到脚本中。把动画从idle1改为idle2，并再次运行场景。为death动画执行相同的操作。测试多个不同的动画，看看它们是如何播放的。

17.5 小结

在本章中，介绍了 Unity 中的动画。你首先了解了动画的基本知识，并且学习了动画和绑定。接着，你从Asset Store 中导入了一个绑定的模型和动画。之后，你把动画应用于模型，并且在运行的场景中看到了它们。在本章最后，你学习了如何通过脚本更改动画。

提示：

动画没有工作

在学完本章后，你可能尝试对自己的一些模型（或者 Asset Store 中的其他模型）制作动画。只需记住：如果正确地应用了动画，但是模型仍然没有移动，那么问题就可能是模型的绑定方式与动画不同。这个提示只是提醒你必须以相同的方式绑定模型和动画，否则很可能会遭受挫败。

17.6 问与答

问：动画可以混合在一起吗？

答：是的，可以。在下一章中将与Unity新增的Mecanim系统一起介绍这个主题。

问：可以把任何动画应用于任何模型吗？

答：仅当它们的绑定方式完全相同时，才可以这样做。否则，动画的行为可能非常怪异，或者根本不会工作。

问：在Unity中可以重新绑定模型吗？

答：是的，在下一章中将介绍如何执行该操作。

17.7 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

17.7.1 问题

1．模型的“骨骼”被称为什么？

2．哪种包装模式用于向前播放动画，然后在循环时向后播放？

3．判断题：Animation组件的Animation属性包含可供模型使用的所有动画。

4．判断题：动画是使用transform.animation.Play( )方法播放的。

17.7.2 答案

1．绑定或绑定方式。

2．Ping Pong 模式。

3．错误。Animation属性是默认的动画，而Animations属性则是动画集合。

4．正确。

17.7.3 练习

在这个练习中，你将创建一个脚本，使用数字键自由地更改动画。在用于第17章（Hour 17）的本书配套资源中可以找到用于这个练习的完整解决方案，其名称为Hour17_Exercise。这个练习使用在前面的练习“向场景中添加士兵”中创建的场景。

1．打开以前创建的场景，确保idle0是士兵的默认动画，并且确保

利用你一直使用的4个动画填充Animation组件的Animations属性。

2．创建一个名为SoldierScript的新脚本，并把它附加到当前位于场景中的士兵模型上。然后把以下代码添加到脚本的Start( )方法中：

```
void Start(){
  if(Input.GetKeyDown(KeyCode.Alpha1))
    transform.animation.Play("idle0");
  else if(Input.GetKeyDown(KeyCode.Alpha2))
    transform.animation.Play("idle1");
  else if(Input.GetKeyDown(KeyCode.Alpha3))
    transform.animation.Play("idle2");
  else if(Input.GetKeyDown(KeyCode.Alpha4))
    transform.animation.Play("death");
}
```

3．运行场景，并且注意到角色正在循环播放idle0动画。按下“2”键，并且注意到动画改变了。试验“1”～“4”键，看看它们如何改变动画。注意：小键盘上的键将不起作用，因为它被明确地编程为使用字母行上面的数字键。

第18章 动画器

在本章中你将学到：

动画器的基本知识；

怎样创建动画器；

怎样利用脚本管理动画器。

在本章中，你将采用以前学过的关于动画的知识，并把它用于Unity新的Mecanim动画系统和动画器。你首先将了解动画器以及它们如何工作。接着，将探讨如何在 Unity 中进行绑定或者更改模型的绑定方式。之后，将创建一个动画器并配置它。在本章最后，将通过脚本操纵动画器，构建一个交互式的演示。

注意：

TRY IT YOURSELF

由于动画器的复杂性，本章将作为一个大的“Try It Yourself”。这意味着你应该遵循并完成书中介绍的所有步骤。这样，在开始组织内容时，就可以确信项目做好了准备。本书中有几章可以漫不经心地阅读，但是本章不在其列。

18.1 动画器的基本知识

在上一章中，你学习了手动控制各个动画。使用该系统，现在可以执行一些高级处理，比如过渡和混合。不过，这样操作比较麻烦并且乏味无趣。在本章中，你开始使用 Unity 新的动画系统，它被称为Mecanim。这个系统利用一种称为动画器（animator）的资源，它应用了多个动画和过渡。然后把动画器应用于模型，以使之运动。这个系统的美妙之处，是可以创建单个动画器，并把它应用于多个不同的模型，以及使它们都以类似的方式运动。不过，在可以构建动画器之前，需要确保模型正确地进行了绑定，并且正确地设置了动画。

18.1.1 绑定模型

回忆前一章可知，必须以完全相同的方式绑定模型和动画，才能使它们正确地工作。这意味着要使为一个模型制作的动画在另一个不同的模型上工作将非常困难。利用新的Mecanim系统，可以在编辑器中重新映射它们的绑定方式，而无需使用任何3D建模工具。结果就是为一个Mecanim模型制作的任何动画将可用于在Unity中重新映射了其绑定方式的任何模型。现在，动画器可以产生大量的动画库，它们可以应用于使用许多不同绑定的大量模型。

模型的绑定是在 Inspector 视图中完成的。对于本节，将使用一个称为 Jack 的模型，它是由才华横溢的 Matt Muzzy 制作的（http://www.mattmuzzy.com/）。在名为 Jack 的文件夹中可以找到这个模型，该文件夹位于用于第18章（Hour 18）的本书配套资源中。Jack模型已经带有一种绑定方式，将需要在 Unity 中配置它以使用动画器。

要导入和配置模型，可以遵循下面这些步骤。

（1）在Unity中，创建一个名为Models的文件夹，并从本书配套资源中把Jack文件夹拖入其中。

（2）在Jack文件夹中查找名为Jack1的模型，并选取它。在Inspector中，单击Animations选项卡，并取消选择Import Animation 属性。然后单击Apply按钮。

（3）在Rig 选项卡上，把动画类型改为Humanoid。这将导致Avatar Definition 属性出现，可以在这里执行绑定映射。查看图18.1，确保你的属性与之匹配，然后单击Apply按钮。

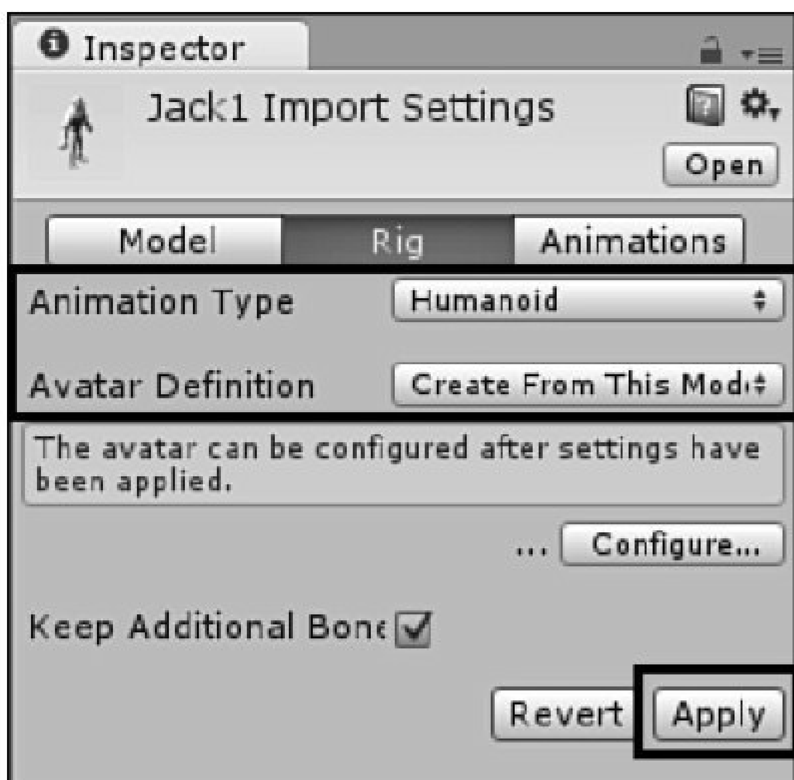


图18.1 绑定设置

（4）单击 Configure 按钮，打开绑定编辑器。如果提示你保存场景，可以继续前进并执行该操作。在绑定编辑器中，将会看到一个站立的模型，它称为所谓的T-Pose，如图18.2所示。

（5）确保所有的关节都是绿色，如图18.3所示，并单击Done按

钮。如果它们不全是绿色，可以参见18.1.2节。

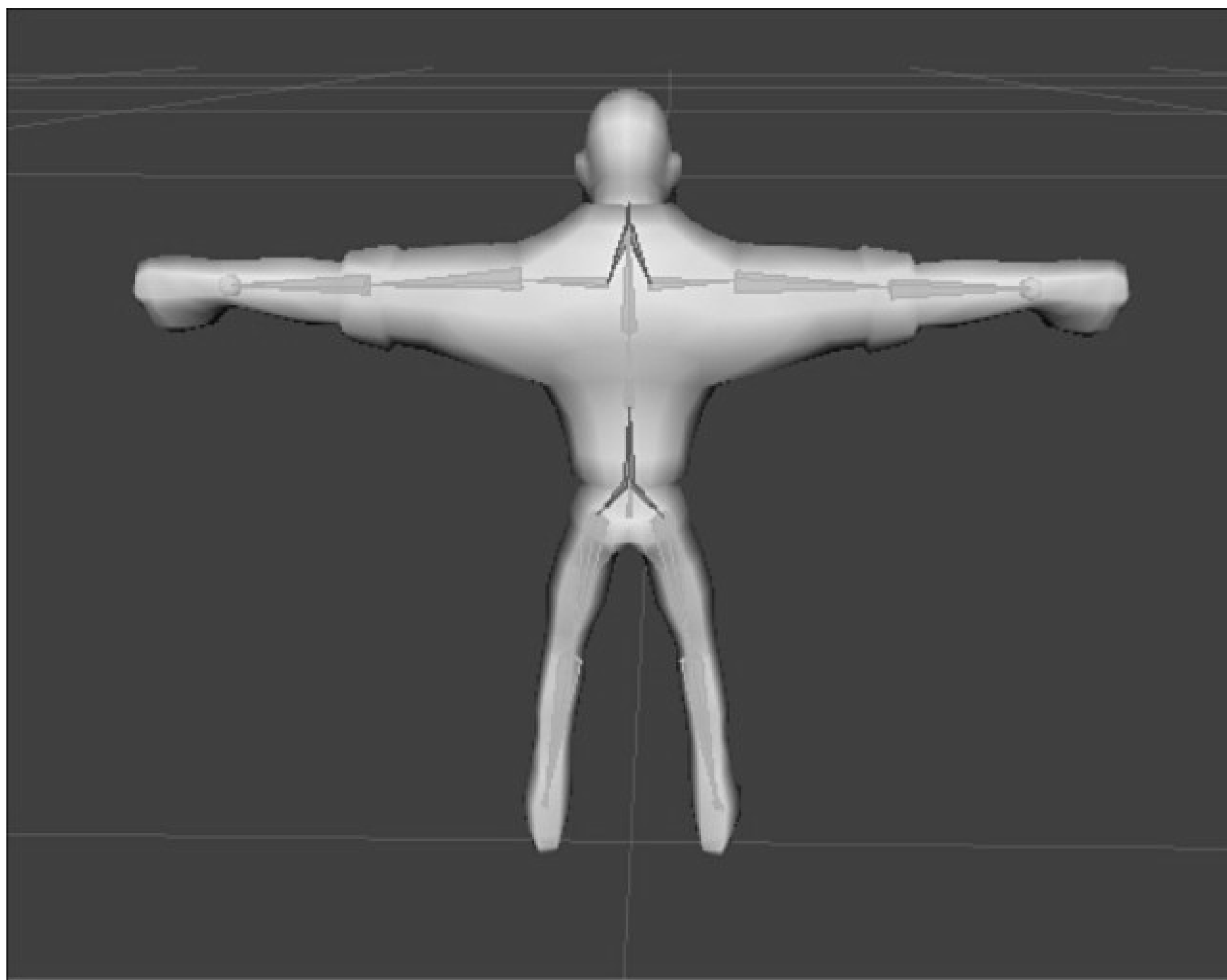
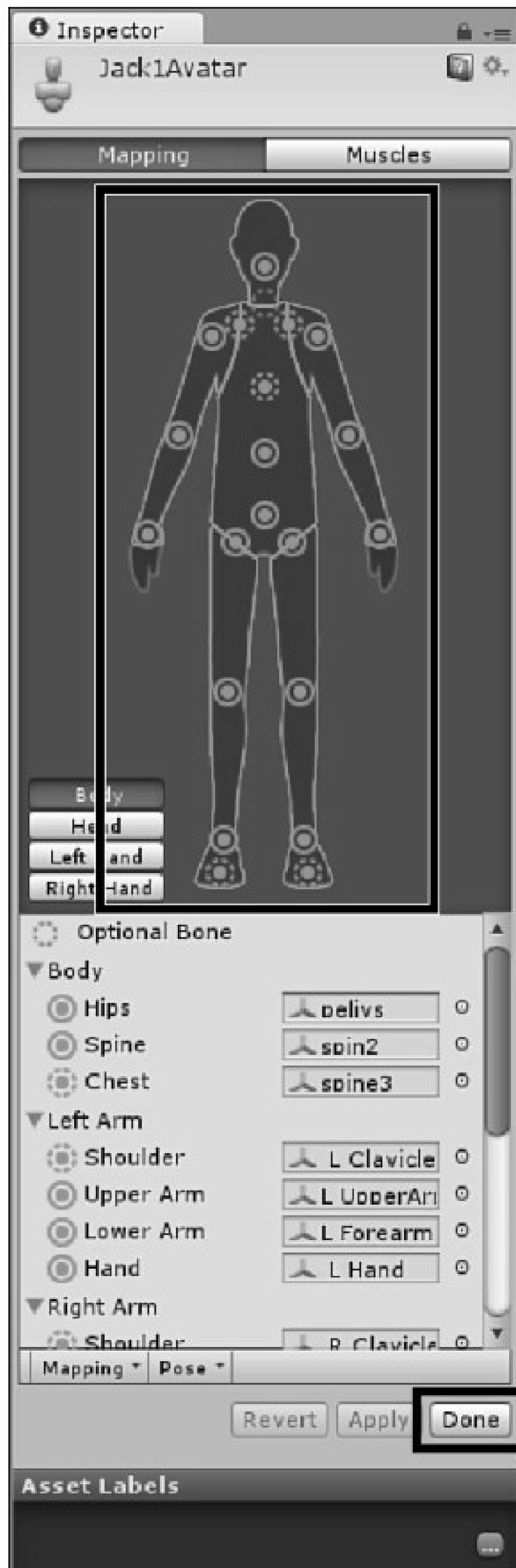


图18.2 T-Pose中的Jack



Inspector
Jack1Avatar
Mapping
Muscles
Body
Head
Left Hand
Right Hand
Optional Bone
Body
Hips
pelivs
Spine
spin2
Chest
spine3
Left Arm
Shoulder
L Clavicle
Upper Arm
Lower Arm
L Forearm
Hand
L Hand
Right Arm
Mapping
Pose
Revert
Apply
Done
Asset Labels

图18.3 成功绑定的模型

模型Jack现在就做好了准备。

18.1.2 死亡的红色绑定

顺利的话，前面的步骤将与计划的完全一样，并且所有的一切都立即会显示为绿色。不过，并非总是如此。Unity有时需要一些帮助，猜测一个模型是如何绑定的。图18.4显示了一个不正确地绑定的模型。在这种情况下，脊椎关节是不正确地匹配的。为了修正它，可以单击Mapping > Automap 命令。这将导致Unity正确地分析模型并修正问题。如果自动映射不适用于特定的模型，可能需要手动查找和映射关节。

有时，模型的绑定将正确地映射，但是有一些关节显示为红色，如图18.5所示。这是姿势限制的结果，可以通过单击Pose > Enforce T-Pose命令加以解决。

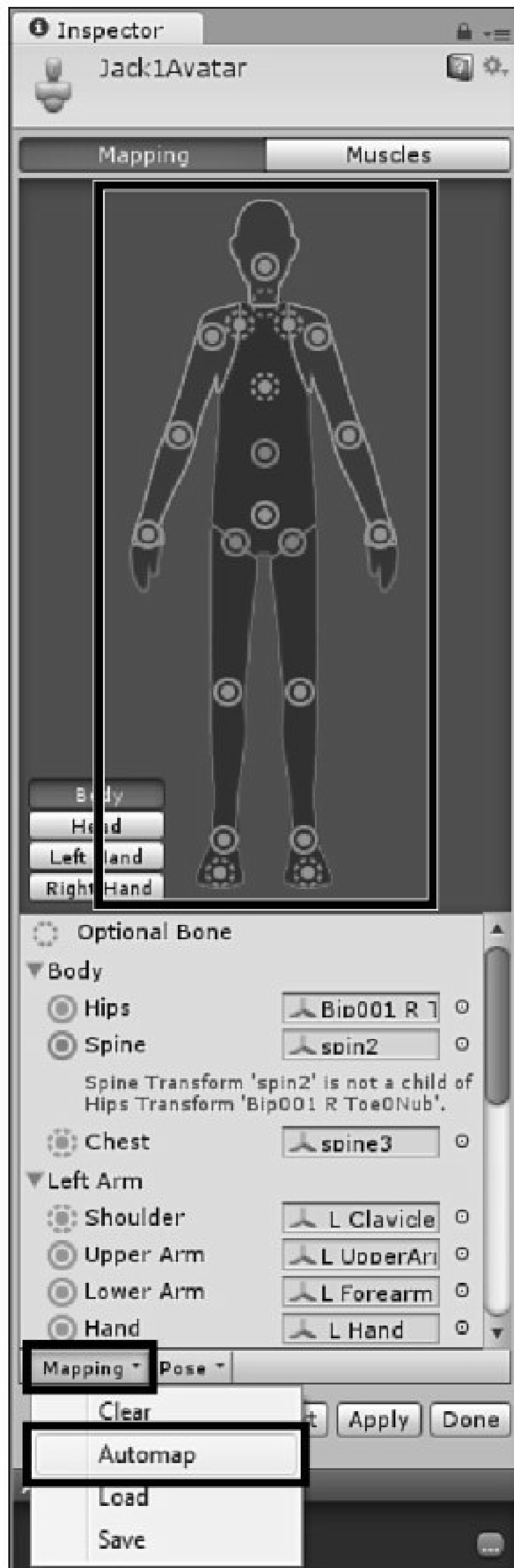

Inspector
Jack1Avatar
Mapping
Muscles
Body
Head
Left Hand
Right Hand
Optional Bone
Body
Hips
Bip001 R T
Spine
spin2
Spine Transform 'spin2' is not a child of Hips Transform 'Bip001 R Toe0Nub'.
Chest
spine3
Left Arm
Shoulder
L Clavicle
Upper Arm
L UpperArı
Lower Arm
L Forearm
Hand
L Hand
Mapping
Pose
Clear
Automap
Load
Save
Apply
Done

图18.4 不正确地映射的模型

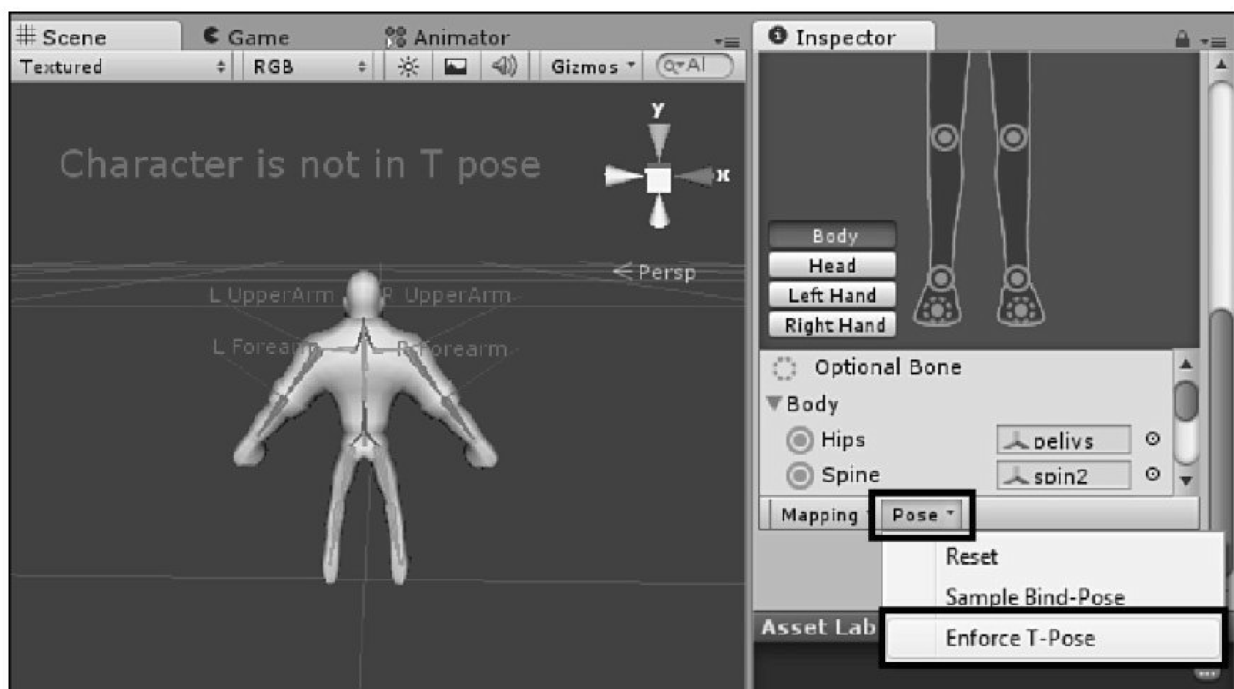


图18.5 强制执行T-Pose

这些简单的方法将修正模型的大多数（如果不是全部）绑定问题。

18.1.3 准备动画

对于本章，将使用一组Mecanim动画。这些动画是由Unity在它们的Mecanim演示中提供的。不过，出于节省时间考虑，可以在用于第18章（Hour 18）的本书配套资源中找到一个名为Animations的文件夹，在该文件夹中可以找到这些动画文件。如果查看该文件夹，将会看到动画实际上是.fbx文件，这是因为动画本身位于它们默认的模型内。不过，不要担心，我们将能够在Unity内修改和提取它们。

必须以你想要的方式明确配置每个动画。例如，你需要确保步行动画适当地循环播放，使得过渡没有任何明显的缝隙。在本节中，你将仔细检查每个动画并准备它。首先将从本书配套资源中把Animations文件夹拖到Unity编辑器中。你将处理3个动画：Idles、WalkForward和

WalkForwardTurns。此外，这3个动画还都需要独特地进行设置。

1．空闲动画

要设置空闲动画，可以遵循下面这些步骤。

（1）在Animations文件夹中选择Idles.fbx文件。在Inspector视图中，选择Rig选项卡。把动画类型改为Humanoid，并且像以前对Jack模型所做的那样准确配置绑定。你可能需要回过头去参考那一节内容，并重复执行其中的步骤。

（2）一旦配置了绑定，就可以单击Inspector中的Animations选项卡。你在这里唯一需要做的是检查Loop Pose 属性。确保你的设置为如图18.6 所示的样子。

（3）动画本身现在应该正确地进行了配置，通过展开Idles.fbx文件可以找到它，如图18.7所示。一定要记住如何访问那个动画，模型本身无关紧要，你想要的是动画。

注意：

红灯、绿灯

你可能注意到了动画设置中存在的绿色圆圈，如图 18.6 所示，它们是非常漂亮的小工具，用于指定动画是否排列好了。圆圈是绿色的事实意味着它们将无缝地循环。如果任何圆圈是黄色的，它就指示动画接近于无缝地循环，但是具有微小的差别。红色圆圈指示动画的开头和结尾根本没有排列好，并且缝隙将非常明显。如果有动画没有排列好，可以更改Start和End属性，找出这样的动画片断。

图18.6 空闲动画的属性

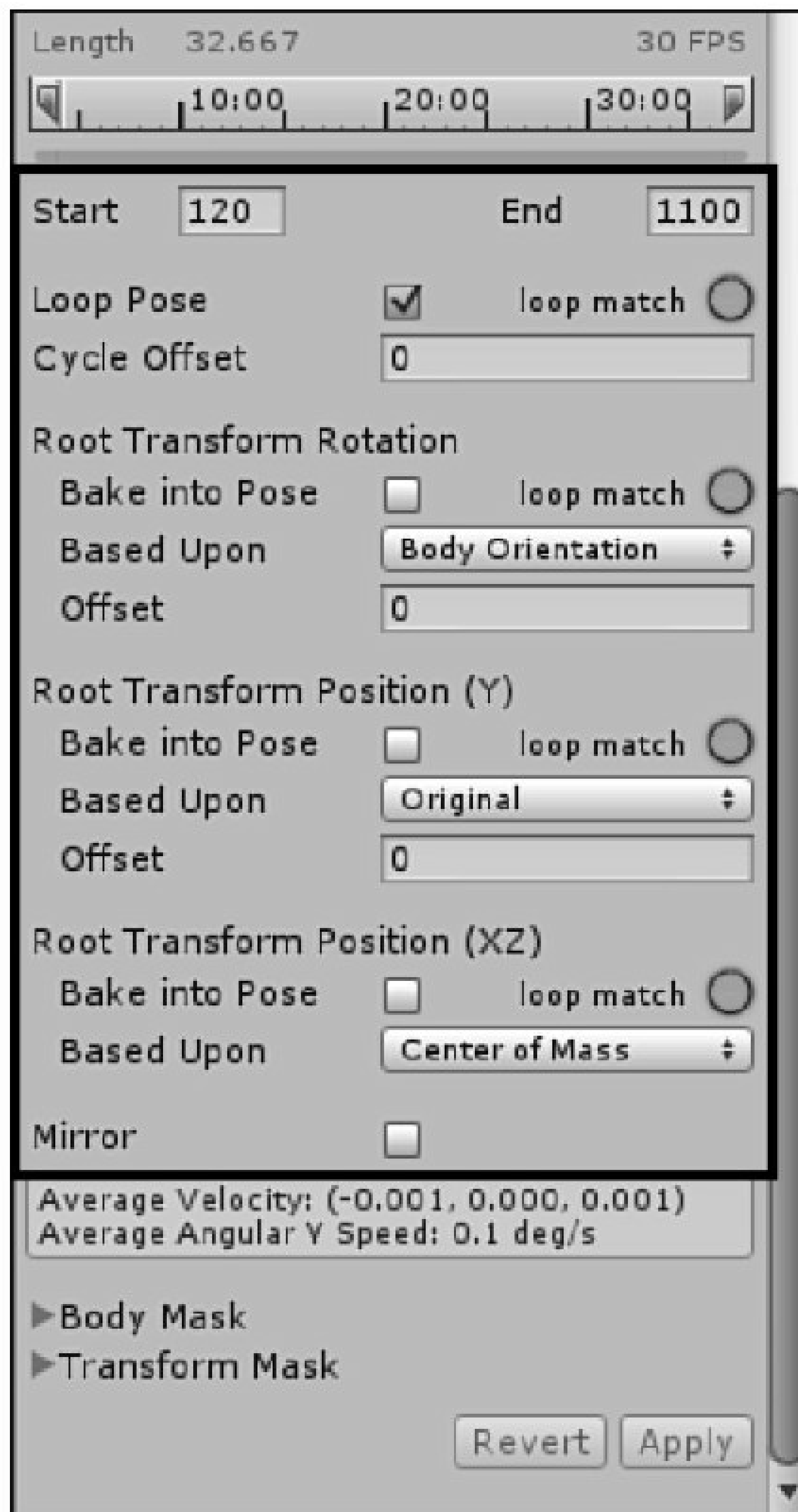


图18.6 空闲动画的属性



图18.7 查找模型中的动画

2．步行动画

要设置步行动画，可以遵循下面这些步骤。

（1）在Animations文件夹中选择WalkForward.fbx文件，并以与空闲动画相同的方式完成绑定。

（2）在Animations选项卡下，应该具有如图18.8所示的设置。你应该注意两件事：第一， Root Transform Position (XZ)旁边具有一个红色圆圈，这很好。它意味着在动画末尾，模型将位于不同的x轴和z轴位置。由于这是一个步行动画，这就是你想要的行为。你应该注意的另一件事是Average Velocity指昧器，应该注意到x 轴速度为−0.034，z 轴速度为1.534。其中z 轴速度是合理的，因为你希望模型向前移动，但是 x 轴速度是一个问题，因为它导致模型在步行时将会横向偏移。需要调整这个设置。

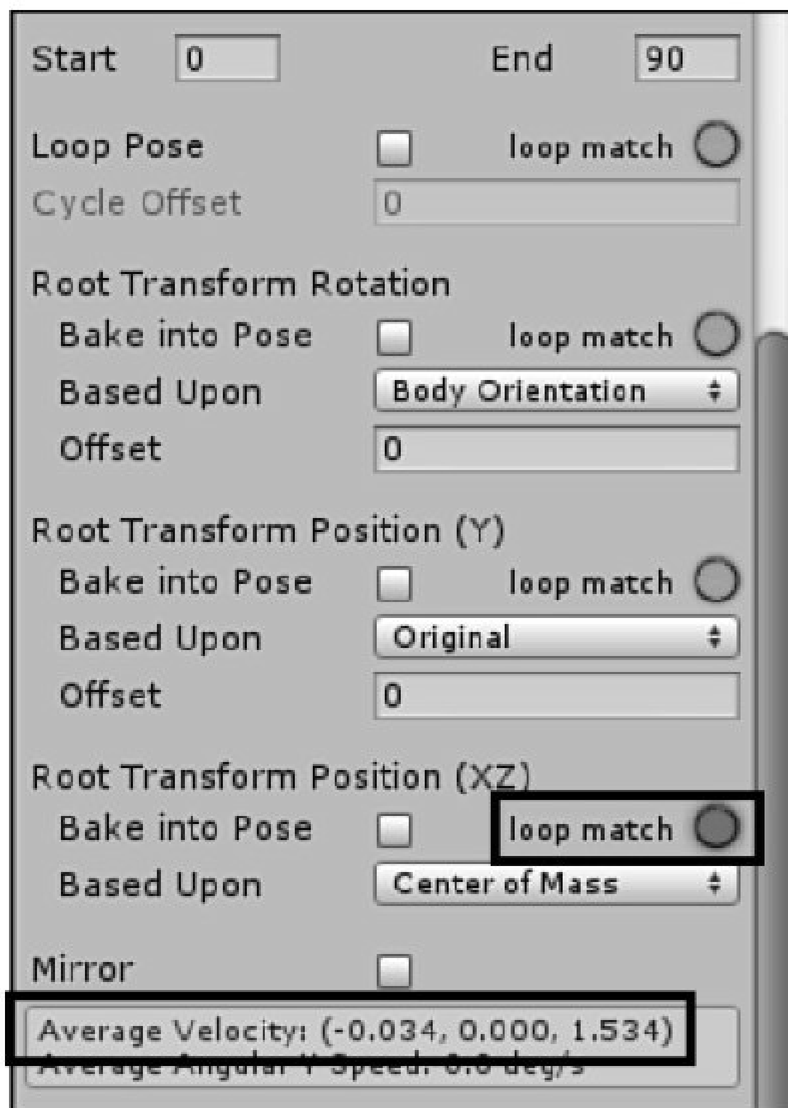


图18.8 步行动画设置

（3）要调整x轴速度，需要检查 Root Transform Rotation 和 Root Transform Position (Y)这两个属性的 Bake into Pose 属性。还需要把 Root Transform Position 偏移量设置为−2.26。最后，你将希望检查 Loop Pose 属性。图18.9包含固定的设置，设置完成后，单击 Apply按钮。

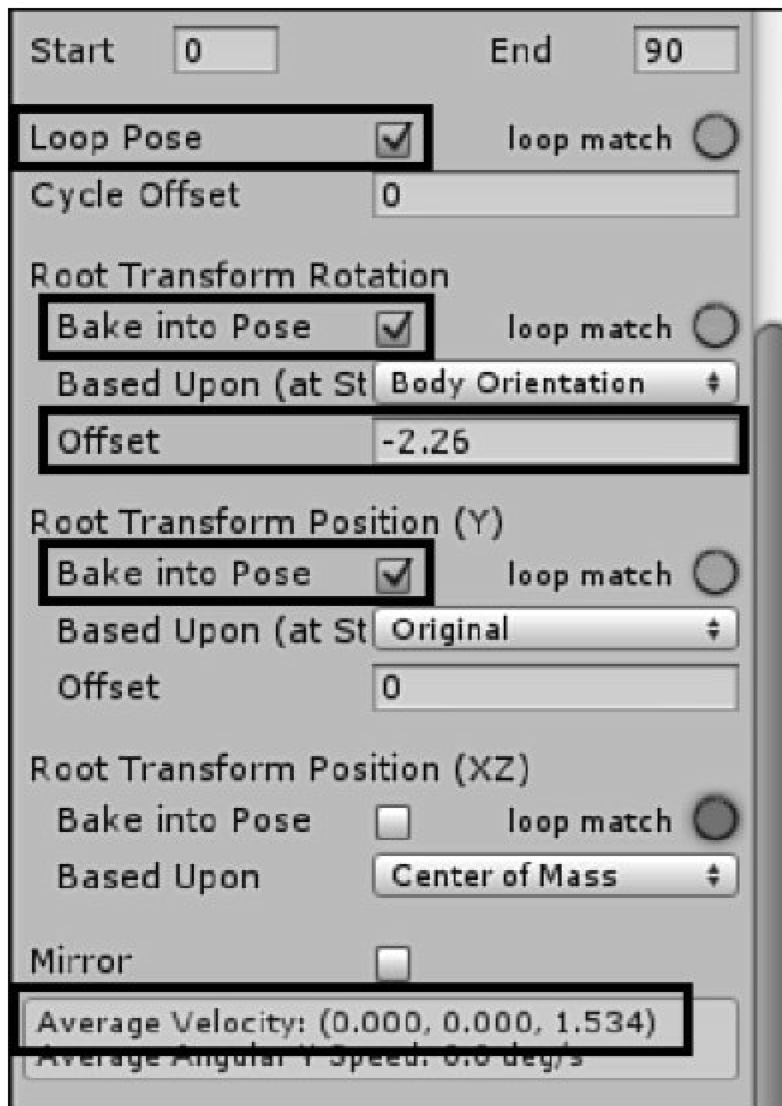


图18.9 固定的步行动画设置

3．步行转弯动画

步行转弯动画允许模型在向前行走时平滑地改变方向。这个动画稍微不同于另外两个动画，因为需要在单个动画记录中制作两个动画。这听起来似乎比实际的更复杂。完成这个动画的步骤如下。

（1）在Animations文件夹中选择WalkForwardTurns.fbx文件，并以与空闲动画相同的方式完成绑定。

（2）默认情况下，将会有一个名为_7_a_U1_M_P_WalkForwardTurnRight的长动画。可以修改它，但是简

单地删除它并重新开始将更容易。在 Clip Name 文本框中输入WalkForwardTurnRight，然后单击加号（+），创建一个新剪辑，如图18.10所示。

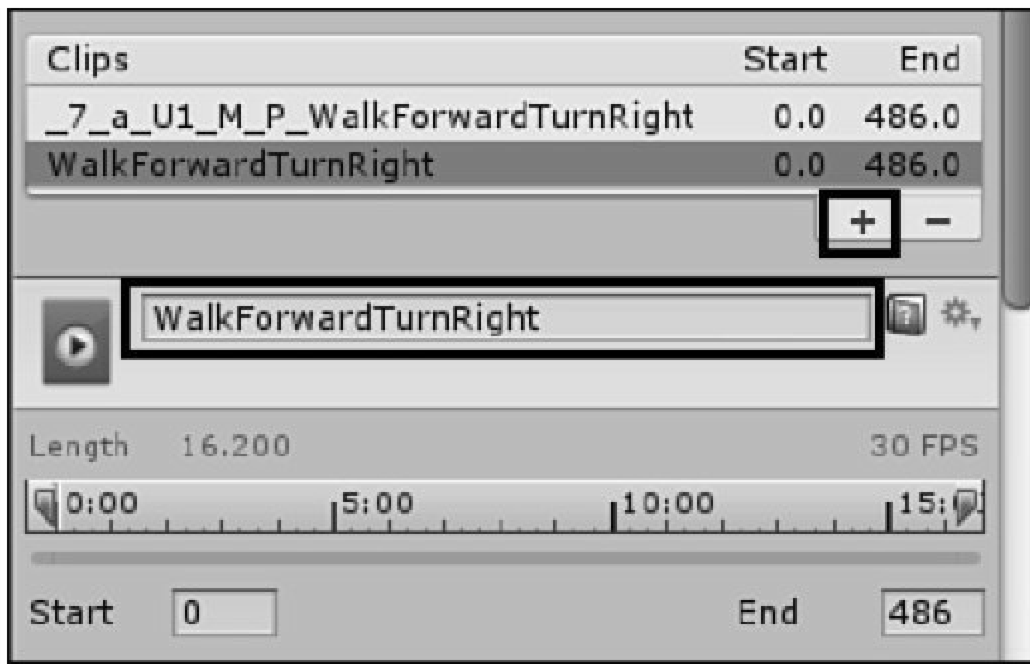
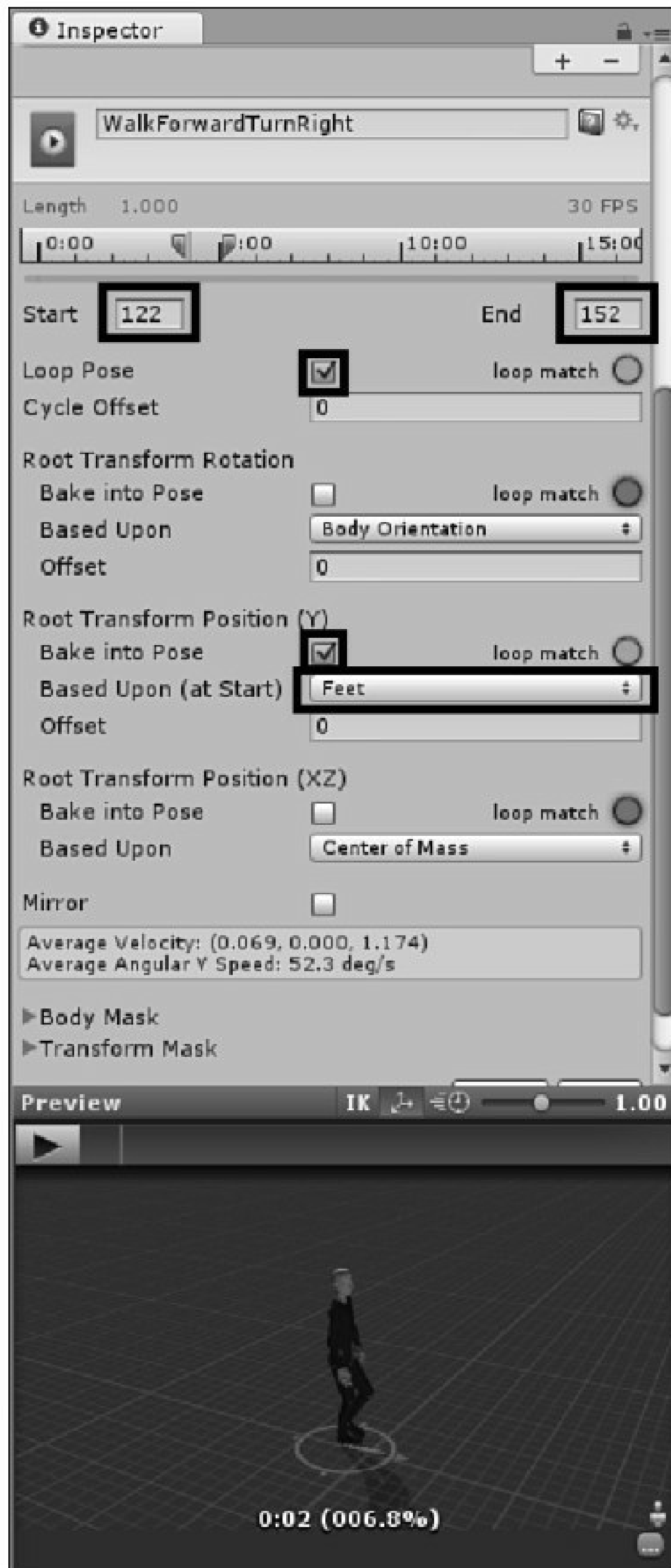


图18.10 添加一个动画剪辑

（3）现在可以删除旧的动画剪辑。选择_7_a_U1_M_P_WalkForwardTurnRight，并单击减号（–）删除它。

（4）选择WalkForwardTurnRight剪辑，把属性设置成如图18.11所示的样子。这将裁短剪辑，并且确保它只包含在右边的圆圈中移动的模型（一定要预览它，查看它是什么样子的）。在做完这些后，可单击Apply按钮。

（5）创建一个WalkForwardTurnLeft动画剪辑，其方法与在第（2）步中制作右转弯剪辑相同。用于WalkForwardTurnLeft剪辑的属性与WalkForwardTurnRight剪辑完全相同，只是需要选中Mirror属性，如图18.12所示。



Inspector
WalkForwardTurnRight
Length 1.000
30 FPS
0:00
5:00
10:00
15:00
Start
122
End
152
Loop Pose
loop match
Cycle Offset
0
Root Transform Rotation
Bake into Pose
loop match
Based Upon
Body Orientation
Offset
0
Root Transform Position (Y)
Bake into Pose
loop match
Based Upon (at Start)
Feet
Offset
0
Root Transform Position (XZ)
Bake into Pose
loop match
Based Upon
Center of Mass
Mirror
Average Velocity: (0.069, 0.000, 1.174)
Average Angular Y Speed: 52.3 deg/s
Body Mask
Transform Mask
Preview
IK
1.00
0:02 (006.8%)

图18.11 右转弯设置

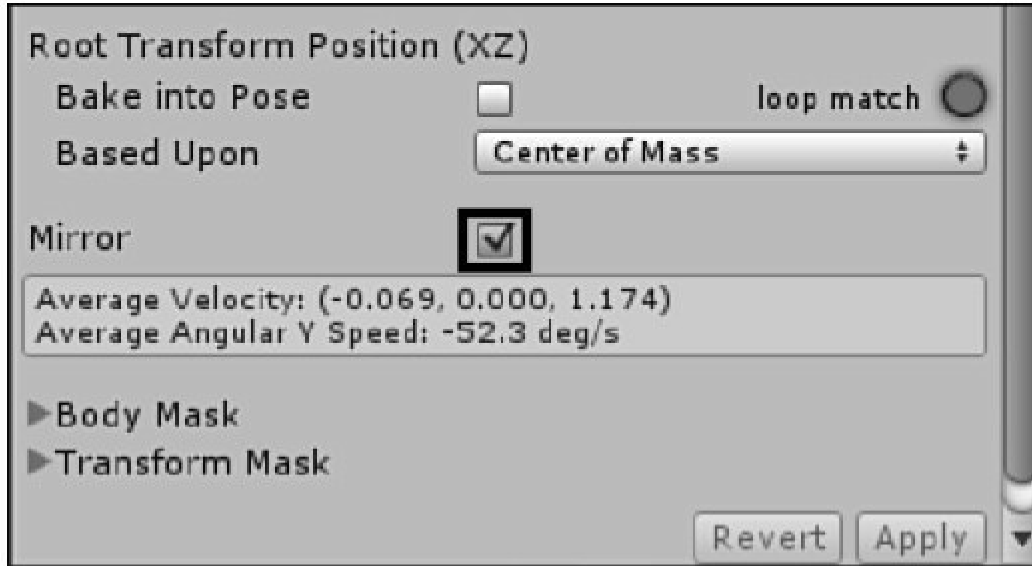


图18.12 镜像动画

此时，就设置了所有的动画，并使它们做好了准备。现在，唯一要做的事情是构建动画。

18.2 创建动画

Unity中的动画是资源，这意味着它们是项目的一部分，并且存在于任何一个场景外面。这样很好，因为这使得可以轻松地反复重用它们。要向项目中添加一个动画，可以在Project视图中右键单击一个文件夹，并且选择Create >Animator Controller 命令。

设置场景

在这个练习中，你将设置一个场景，并为本章的余下内容做好准备。一定要保存在这里创建的场景，因为以后将需要用到它。

（1）如果还没有完成上一节中的工作，就要创建一个新项目，并且完成上一节中的模型和动画准备步骤。

（2）把Jack1模型拖到场景中，并把它定位于(0, 0,−5) 处，然后向场景中添加一个定向灯光。

（3）选取场景中的Jack1 模型。在Jack 文件夹中找到Jack Diffuse1.psd 纹理，并把它拖到场景中的 Jack1 模型上。把 Main Camera 嵌套在 Jack1 模型下（在 Hierarchy 视图中，把 Main Camera 拖到模型上），然后把摄像机定位在(0, 1.5,−1.5)处，并把旋转角度设置为(20, 0, 0)。

（4）创建一个名为Animators 的新文件夹，右键单击该文件夹，并选择Create >Animator Controller 命令。把动画器命名为PlayerAnimator。然后在场景中选取 Jack1，把动画器拖到Inspector视图中的Animator组件的Controller属性上，如图18.13所示。

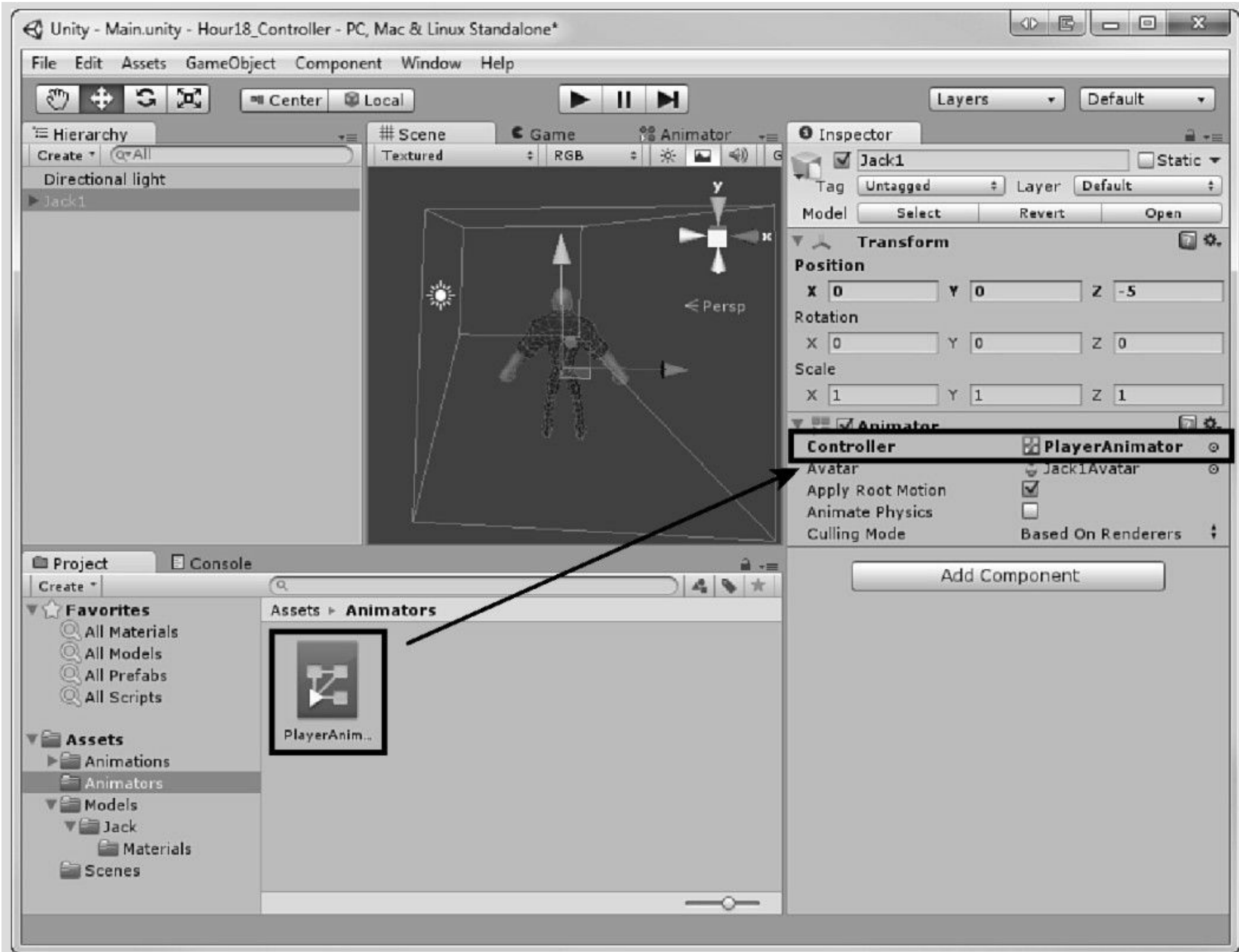


图18.13 给模型添加动画器

（5）向场景中添加一个平面，把它定位于(0, 0, –5)处，并把旋转角度设置为(50, 1, 50)。在用于第18章（Hour 18）的本书配套资源中找到Checker.tga文件，并把它导入到项目中。然后对平面应用该纹理。

（6）运行场景，并且确保一切都看上去正确。注意：此时，没有什么是运动的。

18.2.1 Animator视图

双击一个动画器将调出Animator视图，这个视图就像一幅流程图，允许可视化地创建动画路径和混合。这是Mecanim系统真正强大的地方。图18.14显示了基本的Animator视图。对于新的动画器，这非常朴

素。只有一个基础层，没有参数，还有一个Any State。很快将更详细地讨论它们。

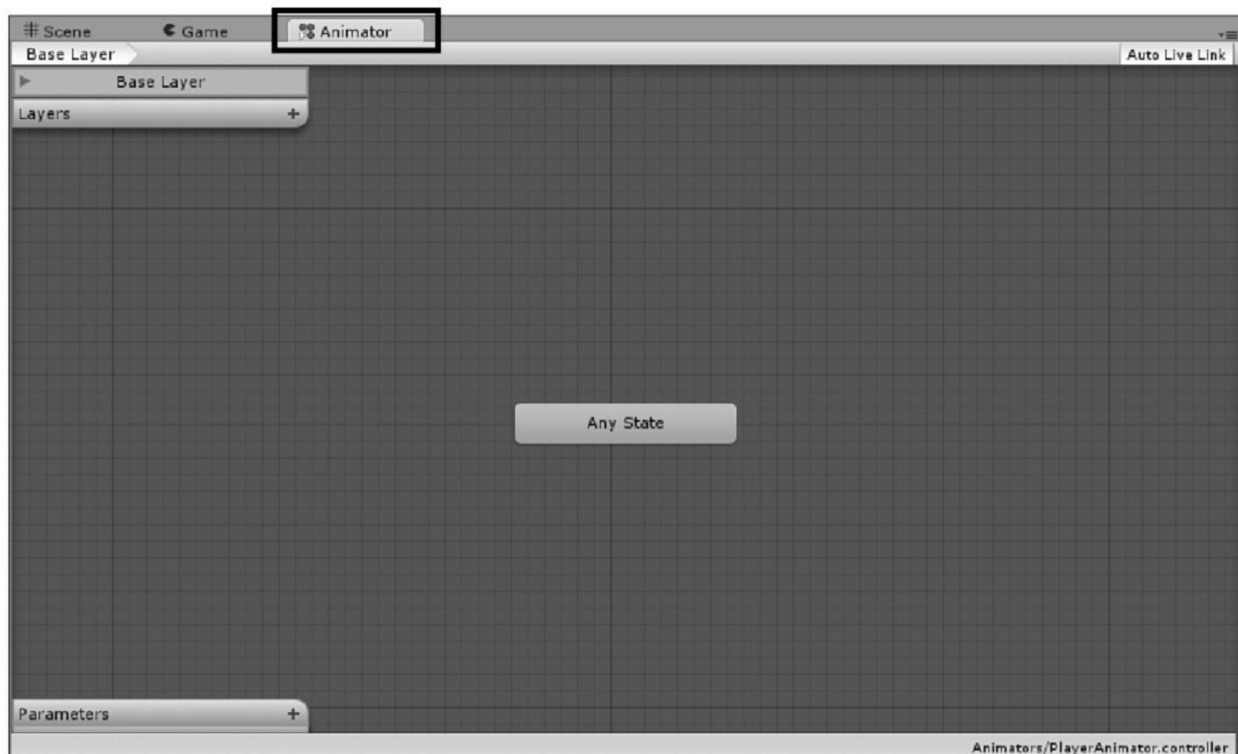


图18.14 Animator视图

18.2.2 空闲动画

你希望应用于 Jack 的第一个动画是空闲动画。既然我们已经完成了整个长长的设置过程，添加这个动画就很简单。你需要找到Idle动画剪辑，它存储在Idles.fbx文件内，如图18.7所示，并把它拖到Animator视图中的动画器上，如图18.15所示。

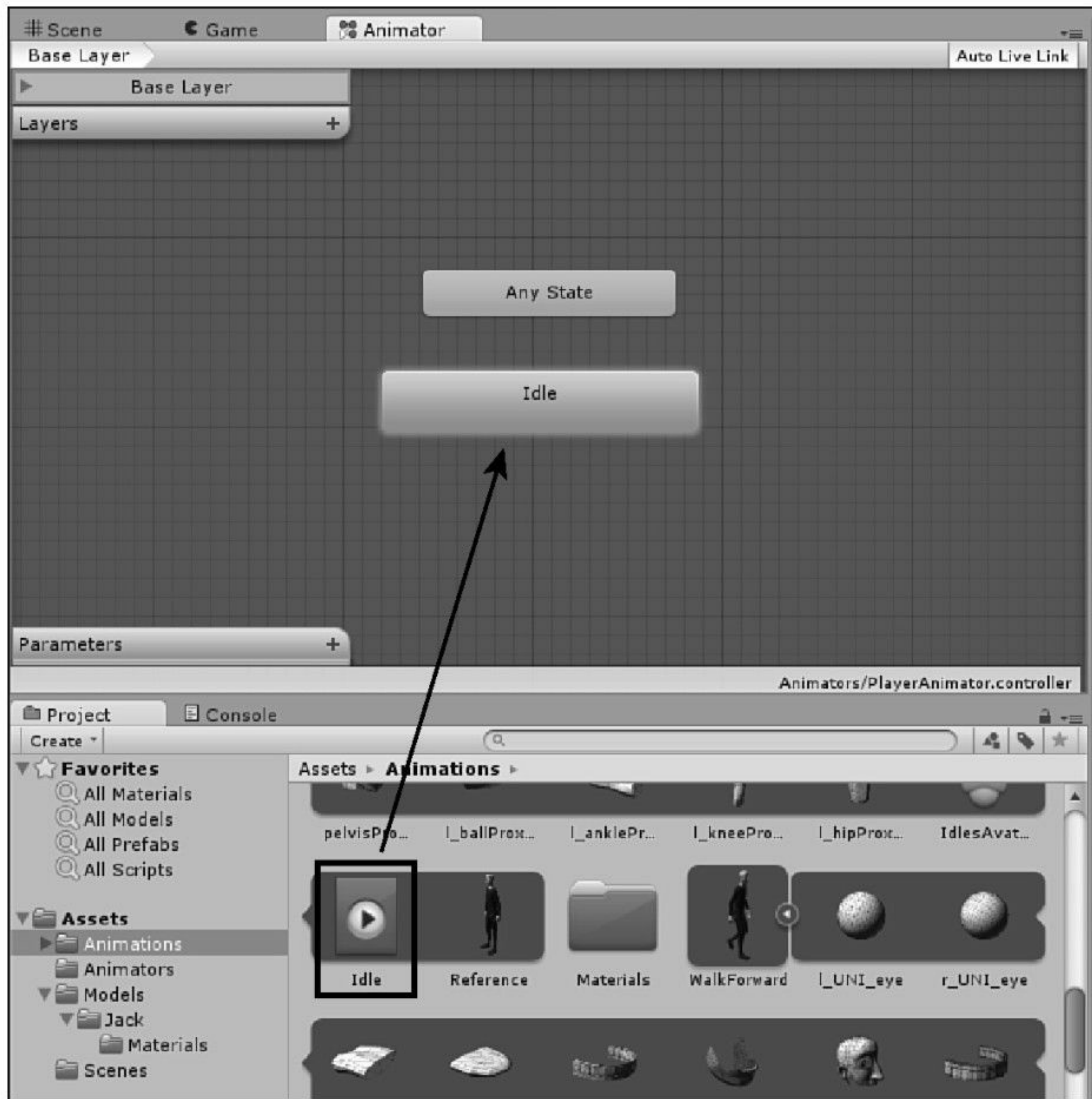


图18.15 应用空闲动画

现在，你可以运行场景，并且观察Jack模型循环经过空闲动画。

18.2.3 参数

参数就像用于动画器的变量，可以在Animator视图中设置它们，然后利用脚本操纵它们。这些参数控制何时过渡和混合动画。要创建一个

参数，只需在Animator视图中的Parameters框中单击加号（+）即可。

添加参数

在这个练习中，将添加两个参数。该练习构建在你迄今为止在本章中处理的项目和场景的基础之上。

（1）确保你已经完成了直到此时的所有步骤。

（2）在Animator视图中，单击加号（+）创建一个新参数。选择一个Float参数，并把它命名为Speed，如图18.16所示。

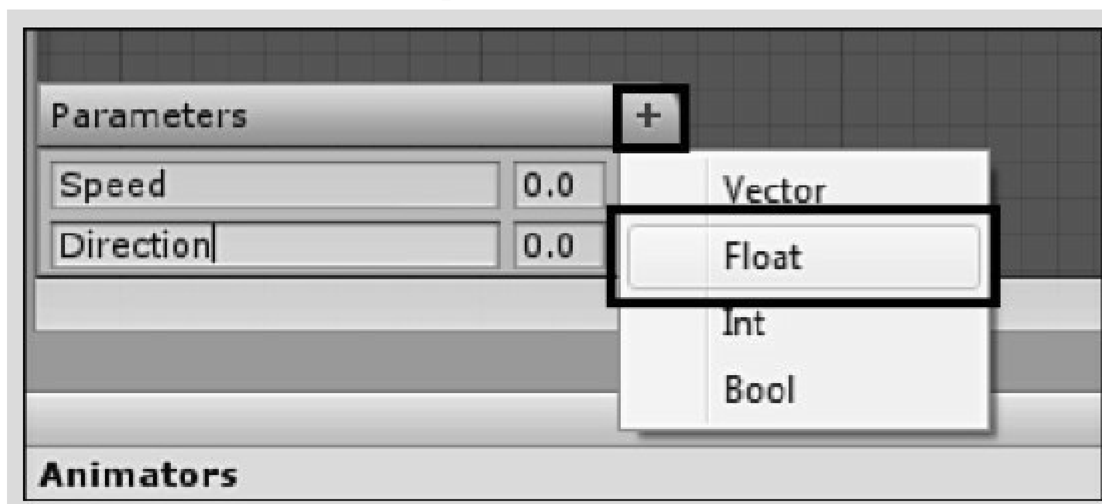


图18.16 添加参数

（3）重复步骤（2），创建一个名为Direction的参数。

18.2.4 状态和混合树

下一步是创建一种新的状态。状态实质上是模型当前所处的状况，它们定义了正在播放的是什么动画。模型Jack具有两种状态：Idle和Walking。Idle已经就位。由于步行状态可以是3个动画中的任意一个，你将希望创建一种使用混合树的状态。

（1）在Animator视图中右键单击一个空白位置，并选择Create State > from New Blend Tree命令。然后在Inspector视图中，把新状态命名为Walking，如图18.17所示。

（2）双击新状态并展开它。在Inspector中，把Parameter属性改为Direction，然后在运动下面单击加号（+），并选择Add Motion Field，添加3 种运动，如图18.18 所示。

（3）把最小值改为–1，如图18.19所示。并把3个步行动画都拖到3个运动字段中（记住：转弯动画剪辑位于WalkForwardTurns.fbx 下）。确保它们的顺序如下：Turn Left、Straight、Turn Right。注意：直行动画将具有一个怪异的名称，因为你永远都不会重命名它。你能够知道它是哪个动画，因为它应该是WalkForward.fbx文件中的唯一一个动画剪辑。

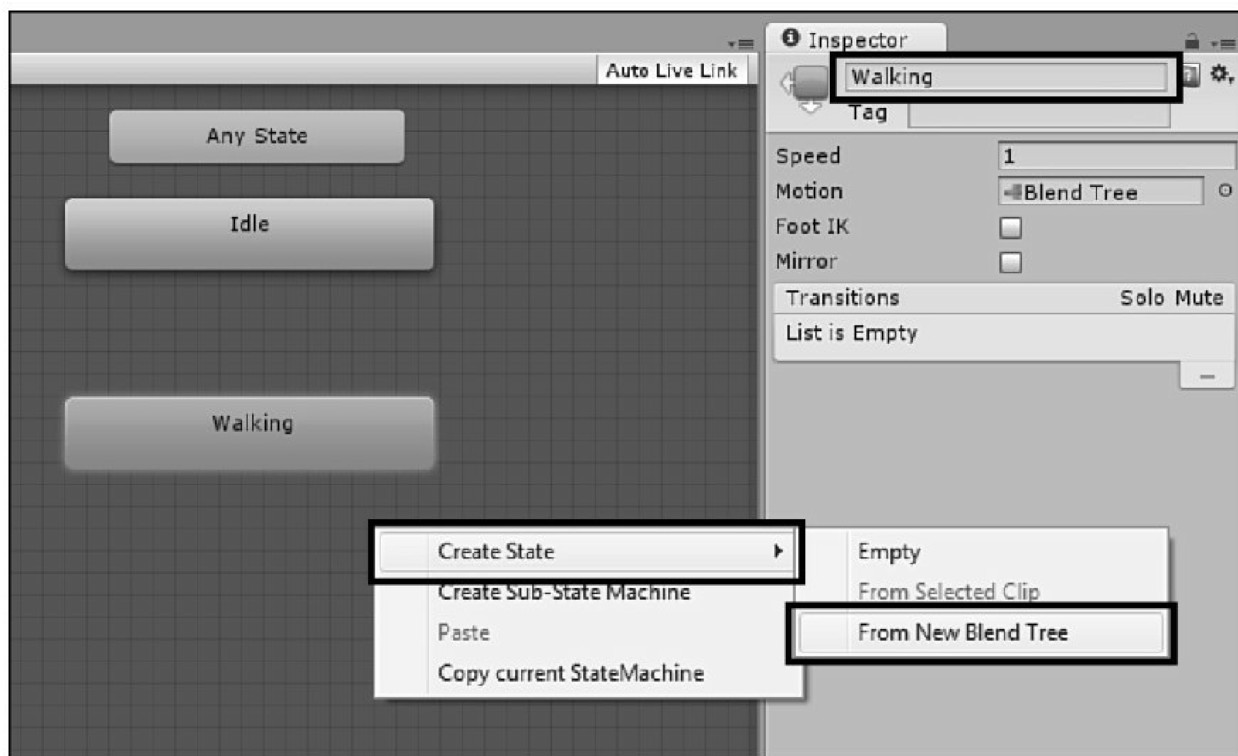


图18.17 创建和命名新状态

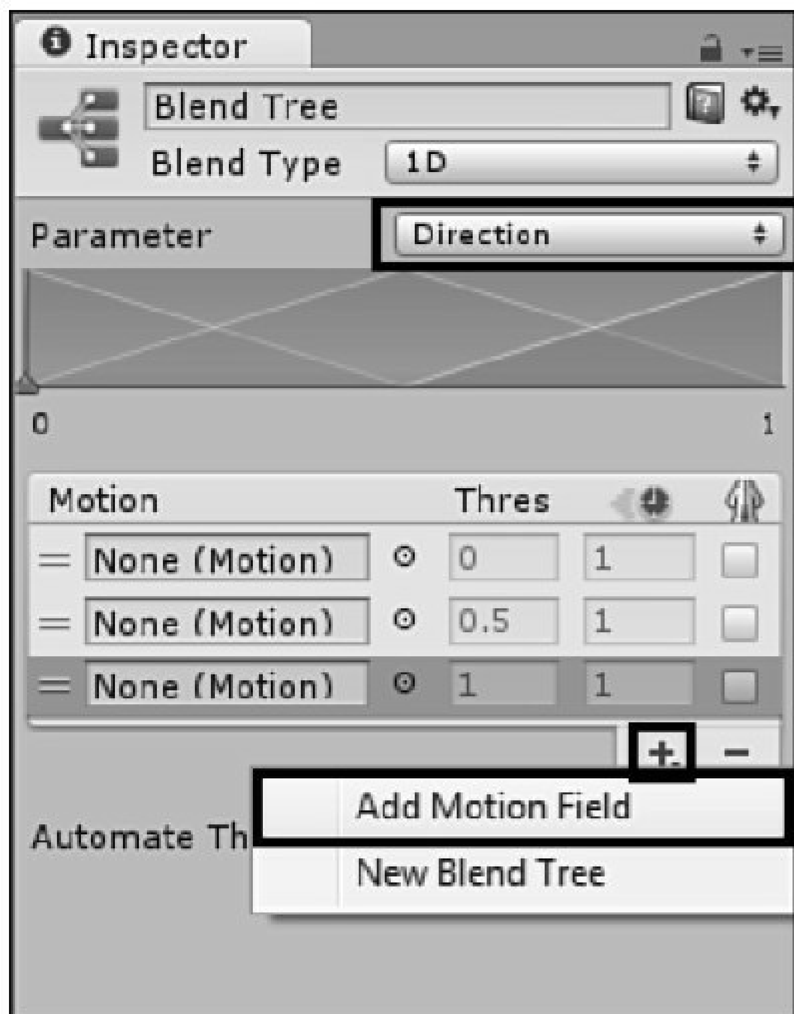


图18.18 添加运动字段

最小值

图18.19 更改最小值并向混合树中添加动画

现在就准备好基于方向参数混合步行动画。可以单击 Animator 视图顶部的 Base Layer导航链接，获得展开的视图，如图18.20所示。

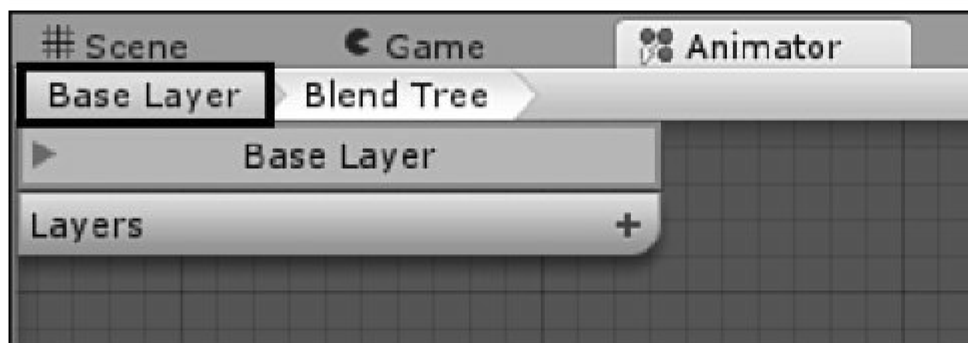


图18.20 导航Animator 视图

18.2.5 过渡

你需要做的最后一件事是：确保动画在完成时告诉动画器如何在空闲动画与步行动画之间进行过渡。你需要设置两种过渡：一种是把动画器从空闲过渡到步行，另一种则是执行相反的过渡。要创建过渡，可以遵循下面这些步骤。

（1）右键单击Idle 状态，并选择Make Transition，这将创建一条跟随鼠标的白线。单击Walking状态，把这两种状态连接起来。

（2）重复第（1）步，只不过这次是把Walking状态连接到Idle状态。

（3）单击 Idle 到 Walking 的过渡上面的箭头以编辑它。把Conditions 设置为其值大于（Greater）0.1的Speed，如图18.21所示。对Walking到Idle的过渡做同样的事情，只不过把条件设置为其值小于（Less Than）0.1的Speed。

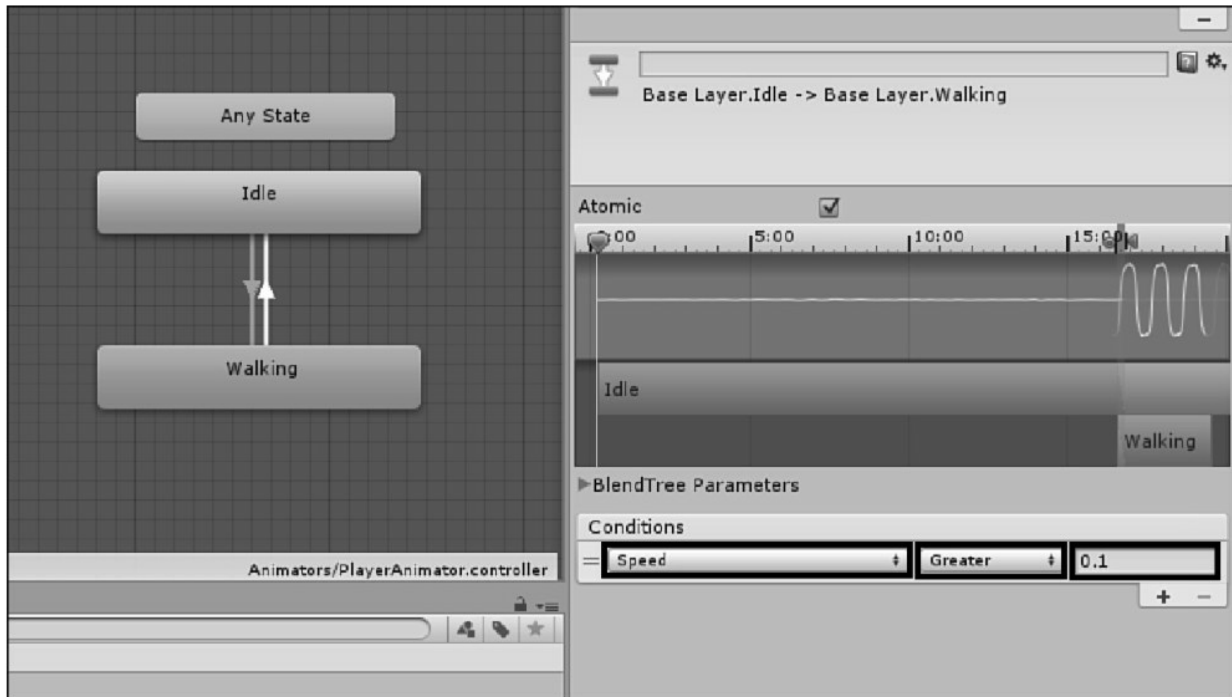


图18.21 修改过渡

动画器完成了。你可能注意到：当运行场景时，没有任何工作的移动动画，这是由于从未改变速度和方向参数。在下一节中，你将学习如何通过脚本更改它们。

18.3 编写动画的脚本

既然已经利用模型、绑定、动画、动画器、过渡和混合树设置好了一切，现在就应该使它们整体上最终具有交互性。幸运的是，实际的脚本组件很简单。大多数困难的工作已经在编辑器中完成了。此时，你只需操纵在动画器中创建的参数，使Jack能够站起来和奔跑。由于你设置的参数是float类型，将需要调用动画器方法：

```
SetFloat(<name> , <value>);
```

最后的润色

这个练习将采用你在学习本章的过程中处理的项目，并添加脚本化组件，以使之完全能够工作。

（1）创建一个名为 Scripts 的新文件夹，并向其中添加一个新脚本。把该脚本命名为AnimationControlScript，并把它附加到场景中的Jack1模型上。

（2）把以下代码添加到脚本中：

```
Animator anim;
void Start () {
   //Get a reference to the animator
   anim = GetComponent<Animator>();
}
void Update () {
   anim.SetFloat("Speed", Input.GetAxis("Vertical"));
   anim.SetFloat("Direction", Input.GetAxis("Horizontal"));
}
```

（3）运行场景，并且注意动画是利用垂直轴和水平轴控制的。

这就行了！如果在添加了这个脚本之后运行场景，就可能注意到一些怪异的事情。不仅Jack通过将经历空闲、步行和转弯这些动作，而且模型也会移动。这是由于两个因素：第一个是选择的动画对它们具有内置的移动，这是通过 Unity 外面的动画完成的，如果还没有实现这一点，将不得不自己编写移动的程序；第二个因素是默认情况下动画器允许动画移动模型，可以在Animator组件的Apply Root Motion 属性中更改它，如图18.22所示。

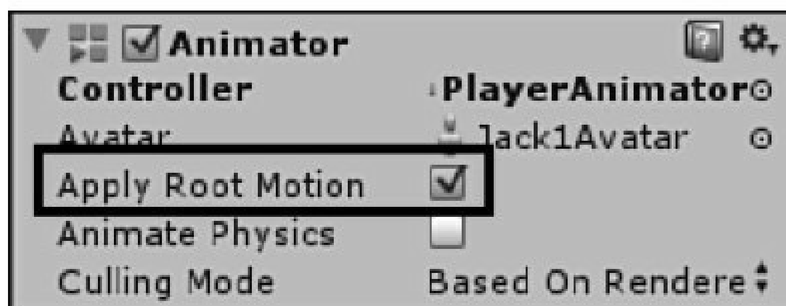


图18.22 Apply Root Motion动画器属性

18.4 小结

在本章中，你学习了在 Unity 中创建动画器。你首先认识了动画器，接着，经历了准备绑定和动画以便用于Mecanim系统的步骤。一旦完成了这些步骤，你就创建了一个动画器，并给它添加了参数、状态、混合树和动画。在本章最后，你学习了如何通过脚本操纵参数来控制动画器。

18.5 问与答

问：这里有许多步骤，Mecanim系统确实优于遗留的系统吗？

答：本章涉及的工作量可能令人畏缩不前。不过，要知道的是，只要稍微熟悉一点，这些步骤就很简单。此外，要记住如果没有Mecanim系统，将不得不针对特定的绑定制作动画。在遗留系统中，不能像你现在这样重新映射绑定。

18.6 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

18.6.1 问题

1．模型必须具有什么姿势，才能在绑定编辑器中正确地进行绑定？

2．存在于动画器内的变量称为什么？

3．哪个方法用于在脚本中设置float参数？

18.6.2 答案

1．T-Pose。

2．参数。

3．SetParam(<name>, <value>)

18.6.3 练习

制作健壮的、高质量的动画系统需要许多信息。在本章中，你看到了用于实现这个目标的一种方式和一组设置。不过，有大量其他的资源可用，而学习是取得成功的首要条件。针对本章的练习是继续研究Mecanim系统。一定要先浏览Unity的关于该系统的文档，可以在Unity的网站上找到它（http://docs.unity3d.com/Documentation/Manual/MecanimAnimationSysten
倘若你想花更多的时间学习该系统，Unity在http://video.unity3d.com/video/7362044/unity-40-mecanim-animation-

tutorial上提供了一个关于该系统的非常好的演示。

第19章 第4款游戏：Gauntlet Runner

在本章中你将学到：

怎样设计Gauntlet Runner 游戏；

怎样构建Gauntlet Runner 游戏世界；

怎样构建Gauntlet Runner 游戏实体；

怎样构建Gauntlet Runner 游戏控制；

怎样进一步改进Gauntlet Runner 游戏。

让我们制作一款游戏！在本章中，你将制作一款3D赛跑游戏，它被适当地命名为Gauntlet Runner。在本章中，首先将开始设计游戏。接着，将重点关注构建游戏世界。一旦完成，将构建游戏实体和控制。在本章最后，将尝试玩游戏，看看可以对哪些地方进行改进。

提示：

完成的项目

一定要遵循本章中的指导，来构建完整的游戏项目。如果你感到迷茫，可以在用于第19章（Hour 19）的本书配套资源中找到游戏的完整副本。如果需要帮助或灵感，可以看一看它。

19.1 设计

在第7章中，你已经学习过设计元素是什么。这一次，你将开始接触到它们。

19.1.1 理念

在这款游戏中，你将扮演一个机器人，跑步通过一条赛道式隧道，并尝试充电，以延长游戏时间。你需要避开一些会降低速度的障碍物。当你用完时间后，游戏就结束了。

19.1.2 规则

游戏规则用于规定玩家如何玩游戏，而且还暗示了对象的一些属性。用于Gauntlet Runner游戏的规则如下。

玩家可以左右移动和跳跃。它们以固定的速度奔跑，并且不能以任何其他的方式移动。

如果玩家撞上某个障碍物，将把它们的速度降低50%，并持续1 秒钟的时间。

如果玩家进行了充电，则将把它们的游戏时间延长1.5 秒钟。

玩家受到屏幕边缘限制。

输掉游戏的条件是时间用完了，没有获胜条件。

19.1.3 需求

这款游戏的需求很简单，如下所示。

一种赛道纹理。

一个玩家模型。

充电装置和障碍物，将在Unity中创建它们。

一个游戏控制器，将在Unity中创建它。

充电装置的粒子效果，将在Unity中创建它。

交互式脚本，将在MonoDevelop 中编写它们。

19.2 游戏世界

这款游戏的游戏世界只是3个立方体，它们被配置成看起来像一条赛道。整个设置相当基本，该游戏还有其他的组件，它们增加了挑战性和趣味。

19.2.1 场景

在设置地面的功能之前，先设置场景，并使之做好准备。要准备场景，可以执行以下操作。

（1）创建一个名为Gauntlet Runner的新项目，然后创建一个名为Scenes的新文件夹，并在该文件夹中把场景另存为Main。

（2）向场景中添加一个定向灯光。

（3）把 Main Camera 定位于(0, 3,−10.7)处，并把旋转角度设置为(33, 0, 0)。然后保存场景。

用于这款游戏的摄像机将固定在某个位置，并悬浮在游戏世界的上方。游戏世界的余下部分将从它底下经过。

19.2.2 地面

这款游戏中的地面本质上将是滚动的，不过，与Captain Blaster 中使用的滚动式背景不同，你实际上将不会滚动任何东西。在下一节中将对此做更多的解释，但是目前只需理解：你只需要创建一个地面对象，即可创建滚动效果。地面本身将包含3个基本的立方体和一种简单的纹理。要创建地面，可以遵循下面这些步骤。

（1）向场景中添加一个立方体，把它命名为Ground，然后把它定

位于(0, 0, 15.5)处，并把缩放比例设置为(10, .5, 50)。向场景中添加另一个名为 Wall 的立方体，然后把它定位于(−5.5, .7, 15.5)处，并把缩放比例设置为(1, 1, 50)。复制墙壁部分，并把新的墙壁项目定位于(5.5, .7, 15.5)处。

（2）创建两个新文件夹：Textures 和 Materials。在用于第19章（Hour 19）的本书配套资源中，找到Checker.tga文件，并把它拖到Textures文件夹中。在Materials文件夹中，创建一种新材质，并把它命名为GroundMaterial。

（3）把GroundMaterial 的纹理设置为你刚才导入的Checker文件。修改材质的Main Color属性，给它提供一种微红色。然后把该材质应用于地面和墙壁。

这就行了！地面相当基本。

19.2.3 滚动地面

你以前见过：通过创建一种背景的两个实例，并以一种“蛙跳”的方式移动它们，即可滚动该背景。在这款游戏中，你将使用一种更聪明的解决方案。每种材质都具有一组纹理偏移量，当选取材质时，可以在Inspector视图中查看它们。你想要做的是，在运行时通过脚本修改这些偏移量。如果把纹理设置为重复（它是默认设置），那么纹理将无缝地循环出现。如果操作正确的话，结果将是无缝地滚动的对象，但是不存在任何实际的移动。要创建这种效果，可以遵循下面这些步骤。

（1）创建一个名为Scripts的新文件夹，然后创建一个名为GroundScript的新脚本，并把该脚本同时附加到地面和墙壁上。

（2）把以下代码添加到脚本中（替换其中已经存在的Update( )方法）：

```
float speed = .5f;
```

```
void Update () {
    float offset = Time.time * speed;
    renderer.material.mainTextureOffset = new Vector2(0, -offset);
}
public void SlowDown()
{
    speed = speed / 2;
}
public void SpeedUp()
{
    speed = speed * 2;
}
```

（3）运行场景，并且注意到赛道滚动。这是一种创建滚动式3D对象的简单、高效的方式。

在前面的脚本中，你可能注意到两个额外的方法：SlowDown 和 SpeedUp。现在不会使用它们，但是以后当玩家撞到障碍物时它们将是必不可少的。图19.1显示了像前面描述的那样设置的运行的场景。

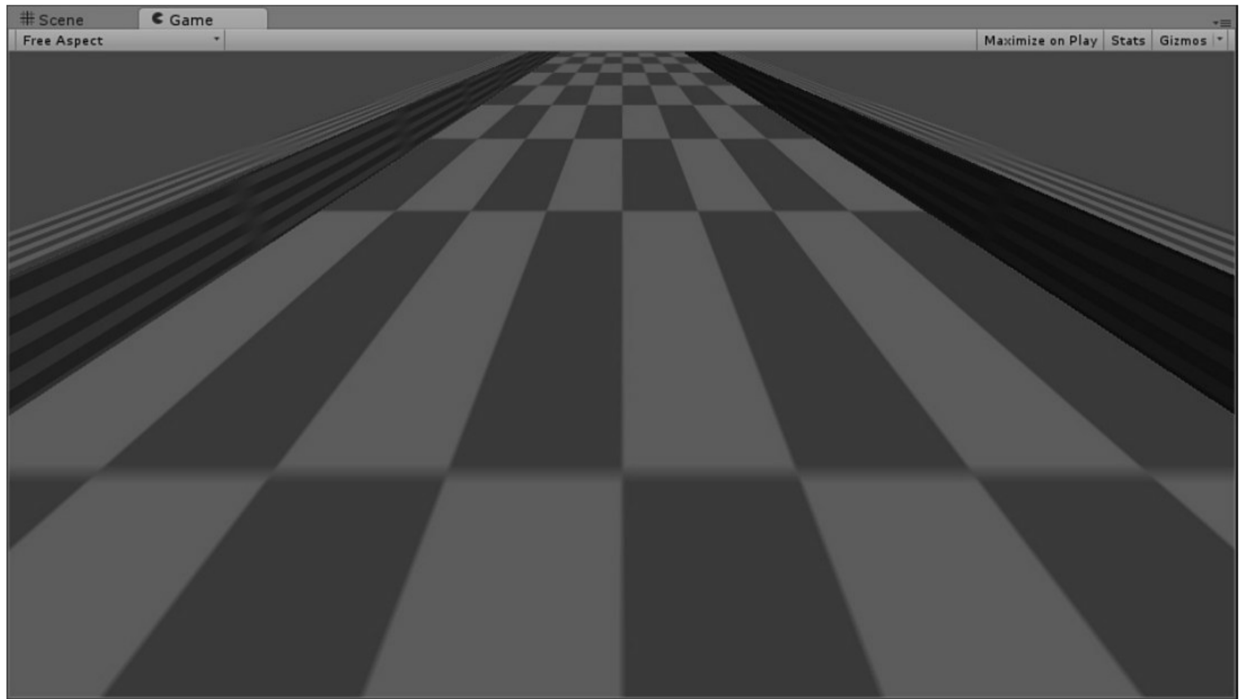


图19.1 运行的赛道

19.3 实体

既然你已经具有一个滚动的游戏世界，现在就应该设置实体。要知道4个主要的实体：玩家、充电装置、障碍物和触发器区域。触发器区域将用于清理使之超越玩家的任何项目，并且无需为这款游戏创建一个复活点。作为替代，你将探索一种处理它的不同方式，即允许游戏控制创建充电装置和障碍物。

19.3.1 充电装置

这款游戏中的充电装置将是简单的球体，并且给它添加了一些效果。你将创建球体，定位它，然后通过它制作一个预设。要创建充电装置，可以遵循下面这些步骤。

（1）向场景中添加一个球体，并把它定位于(0, 1, 42)处。给该球体添加一个刚体，并取消选中Use Gravity。

（2）创建一种名为PowerupMaterial的新材质，并把它设置为黄色。然后把该材质应用于球体。

（3）给球体添加一个点光源（单击Component > Rendering > Light命令），并把它设置为黄色。然后给球体添加一个粒子系统（单击Component > Effects > Particle System命令），然后把粒子的起始颜色设置为黄色，并把起始寿命设置为2.5。

（4）创建一个名为 Prefabs 的新文件夹，在该文件夹中创建一个新的预设，并把它命名为Powerup。从Hierarchy视图中单击球体，并把它拖到预设上。然后从场景中删除球体。

注意：

在把对象放入预设中之前，通过设置对象的位置，可以简单地实例化预设，并且它将出现在那个位置。结果就是将不需要一个复活点。图19.2显示了完成的充电装置。

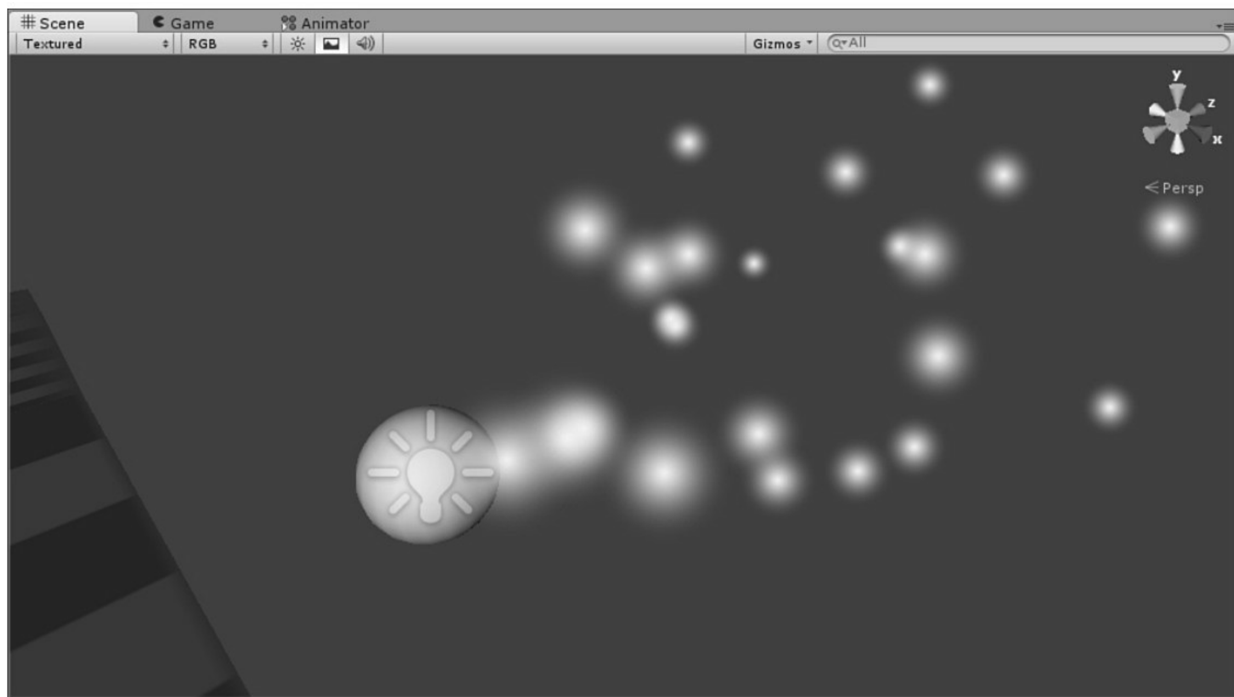


图19.2 充电装置

19.3.2 障碍物

对于这款游戏，通过较小的黑色立方体表示障碍物。玩家可以选择避开它们，或者从它们上面跳过去。要创建障碍物，可以遵循下面这些步骤。

（1）向场景中添加一个立方体，然后把它定位于(0，.4，42)处，并把缩放比例设置为(1，.2，1)。给立方体添加一个刚体，并取消选中Use Gravity。

（2）创建一种名为ObstacleMaterial的新材质，所该材质的颜色设置为黑色，并把它应用于立方体。

（3）创建一个名为 Obstacle 的新预设，从 Hierarchy 视图中把立方

体拖到预设上，然后删除该立方体。

19.3.3 触发器区域

就像前面的游戏中一样，触发器区域用于清理使之超越玩家的任何游戏对象。要创建触发器区域，可以遵循下面这些步骤。

（1）向场景中添加一个立方体，把该立方体命名为TriggerZone，然后把它定位于(0, 1, −20)处，并把缩放比例设置为(10, 1, 1)。

（2）在触发器区域的Box Collider 组件上，选中Is Trigger 属性。

19.3.4 玩家

玩家将是为这款游戏所做的大部分工作。玩家将使用你尚未接触过的两个新动画：跑和跳。你首先将使玩家为Mecanim动画做好准备。

（1）在用于第19章（Hour 19）的本书配套资源中找到名为Robot Kyle的文件夹，这是一个由Unity提供的可免费使用的模型。不过，为了节省在Asset Store上查找它的时间，在这里提供了它。把该文件夹拖到Unity中的Project视图中，以导入它。

（2）在 Robot Kyle 文件夹下的 Model 文件夹中找到并选择 Robot Kyle.fbx 文件。然后在 Inspector 视图中，选择 Animations 选项卡，并取消选择 Import Animation。然后单击Apply按钮。

（3）在Rig选项卡下，把动画类型改为Humanoid。然后单击Apply按钮。

现在应该会在Configure按钮旁边看到一个选中标记，如图19.3所示。如果没有看到它，将需要单击Configure按钮，并且配置绑定。在第18章中可以找到关于这样做的指导（尽管它应该不是必要的）。

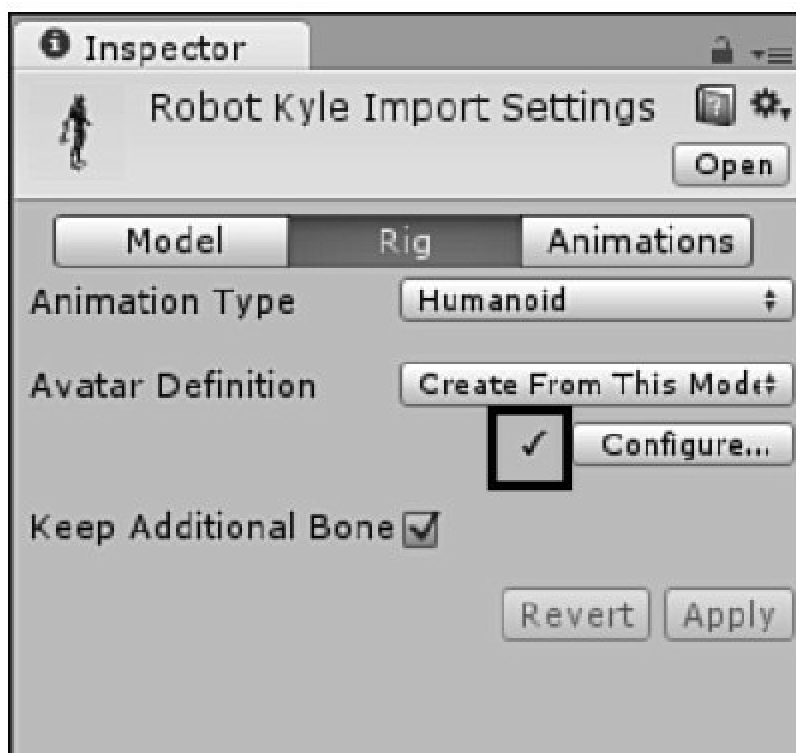


图19.3 绑定设置

现在需要使动画准备好放在动画器中，如下所示。

（1）在用于第19章（Hour 19）的本书配套资源中找到Animations文件夹，并把该文件夹拖到Unity中的Project视图中，以导入它。

（2）在新导入的Animations文件夹中，找到Jump.fbx文件并选择它。在Inspector视图中，单击Rig选项卡，并把动画类型改为Humanoid。然后单击Apply按钮。

（3）在Animations选项卡下，把跳跃动画的属性改为如图19.4所示的样子。注意，Root Transform Rotation属性下面的Offset属性可能需要不同于图中所示的属性，重要的是Average Velocity属性对于x轴具有值0。然后单击Apply按钮。

步，为这个模型校正绑定。在Animations选项卡下，注意有3个剪辑：

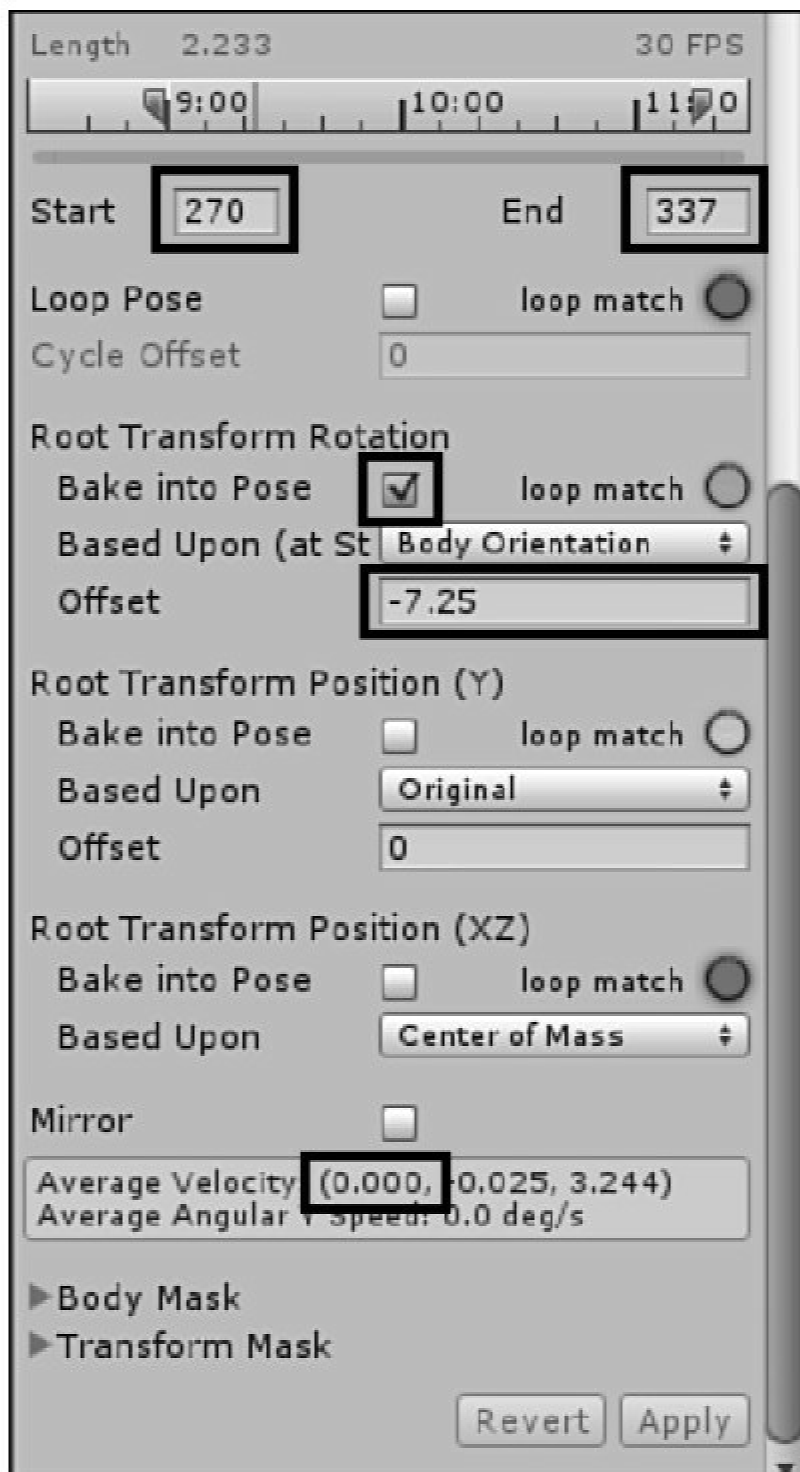


图19.4 用于跳跃动画的属性

（4）在Animations文件夹中选择Runs.fbx文件。再次完成第（2）步，为这个模型校正绑定。在Animations选项卡下，注意有3个剪辑：

RunRight、Run和RunLeft。选择Run，并且确保属性与图19.5 匹配。同样，重要的部分是用于Average Velocity属性的x 轴的值为0。然后单击Apply按钮。

Clips	Start	End
RunRight	0.0	235.0
Run	302.0	319.0
RunLeft	0.0	235.0

+ −

Run

Source Take: Run

Length 0.567 30 FPS

0:00 11:00

Start 302 End 319

Loop Pose ☑ loop match

Cycle Offset 0

Root Transform Rotation

Bake into Pose ☑ loop match

Based Upon (at St: Body Orientation

Offset -4.93

Root Transform Position (Y)

Bake into Pose ☐ loop match

Based Upon: Original

Offset 0

Root Transform Position (XZ)

Bake into Pose ☐ loop match

Based Upon: Center of Mass

Mirror ☐

Average Velocity: (0.000, 0.037, 5.398)
Average Angular Y Speed: 0.0 deg/s

▶Body Mask
▶Transform Mask

图19.5 跑步动画的属性

既然已经准备了动画，就可以开始制作动画器。这将是一个简单的包含两个阶段的动画器，并且无需任何混合树。要准备动画器，可以遵循下面这些步骤。

（1）创建一个名为 Animators 的新文件夹，并在该文件夹中创建一个新的动画器（右键单击并选择Create > Animator Controller 命令），并把它命名为PlayerAnimator。

（2）双击动画器，打开Animator视图。在Animations文件夹中，通过单击它左边的箭头找到Runs.fbx文件。在展开的模型中，找到Run动画剪辑，并把它拖到Animator视图上，如图19.6 所示。单击新创建的Run 状态，然后在Inspector视图中选中Foot IK 属性。

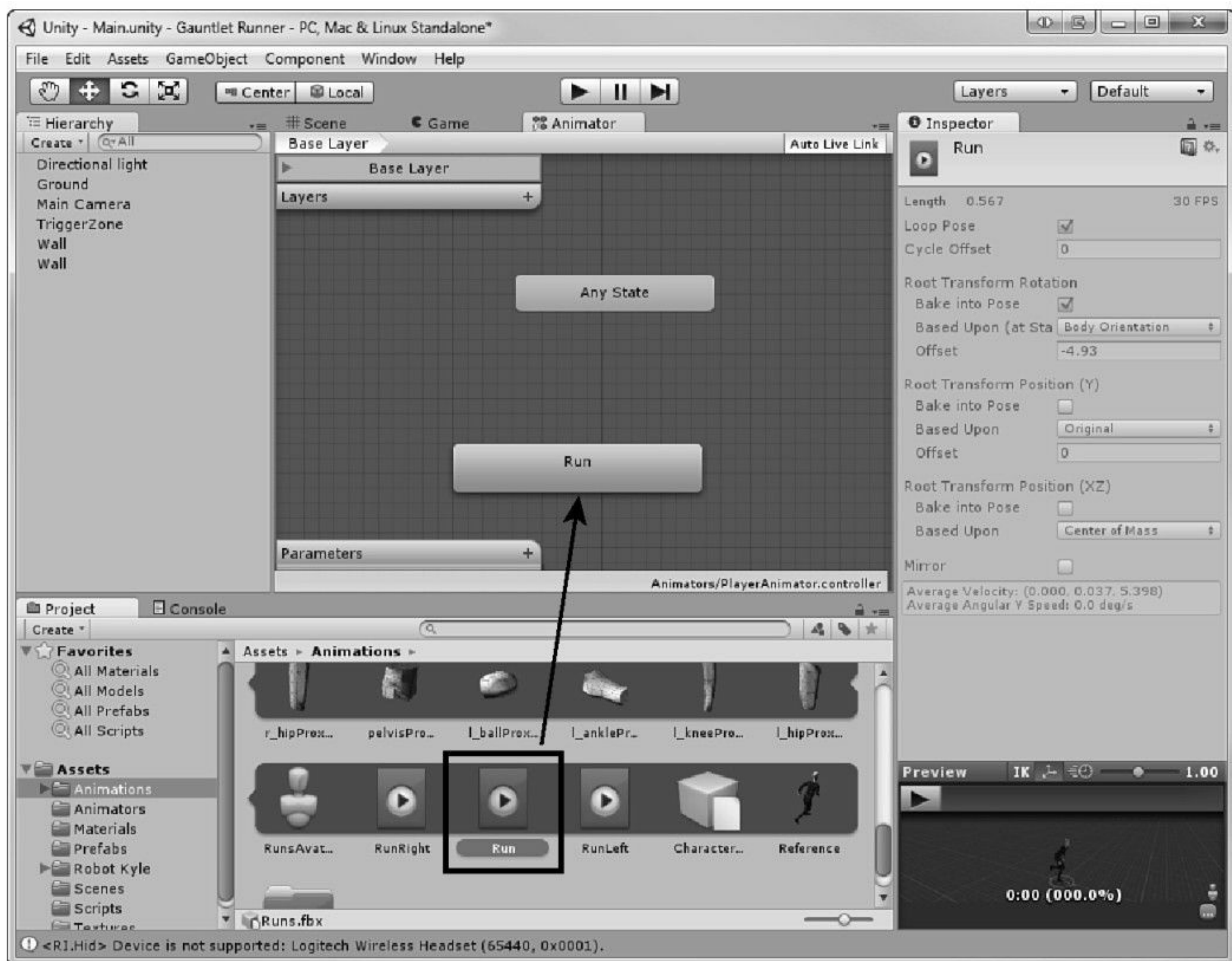


图19.6 添加Run动画剪辑

（3）在Animations文件夹中找到Jump.fbx文件。展开该文件，找到Jump动画剪辑，并把该剪辑拖到Animator视图上。单击新创建的Jump状态，然后在Inspector视图中选中Foot IK属性，并把Speed属性改为1.25。

（4）在Animator视图中单击Parameters框中的加号（+），给动画器添加一个新的参数。该参数应该是一个名为Jumpingr布尔型参数，如图19.7所示。

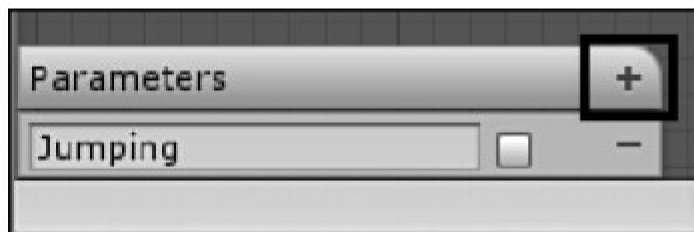


图19.7 添加Jumping 参数

（5）在动画器中右键单击Run 状态，并选择Make Transition。单击Jump 状态，把它们链接在一起。然后右键单击Jump 状态，并选择Make Transition，把它链接回Run 状态。

（6）单击白色箭头，从Run过渡到Jump。在Inspector视图中，把Conditions改为“当Jumping是True时”，如图19.8所示。

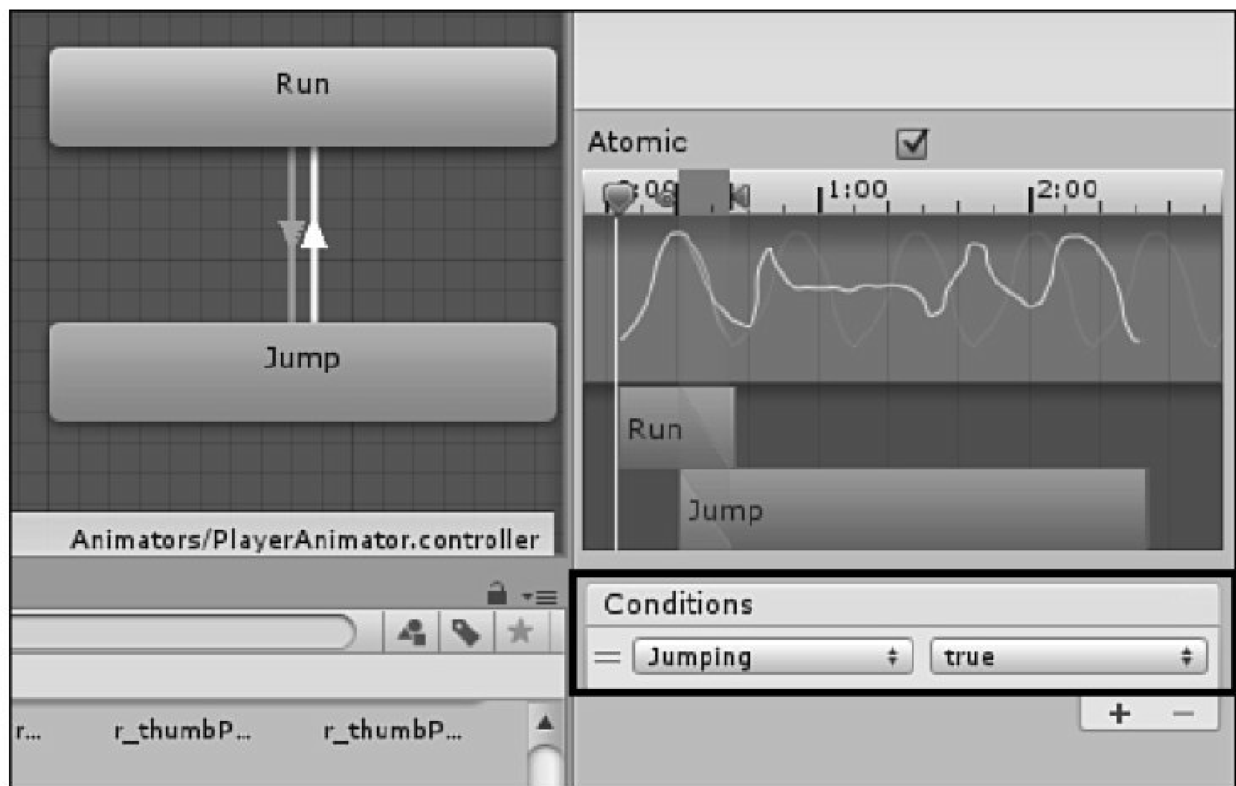


图19.8 Run过渡属性

（7）单击白色箭头，从Jump过渡到Run。确保Inspector视图中的属性为如图19.9所示的样子。

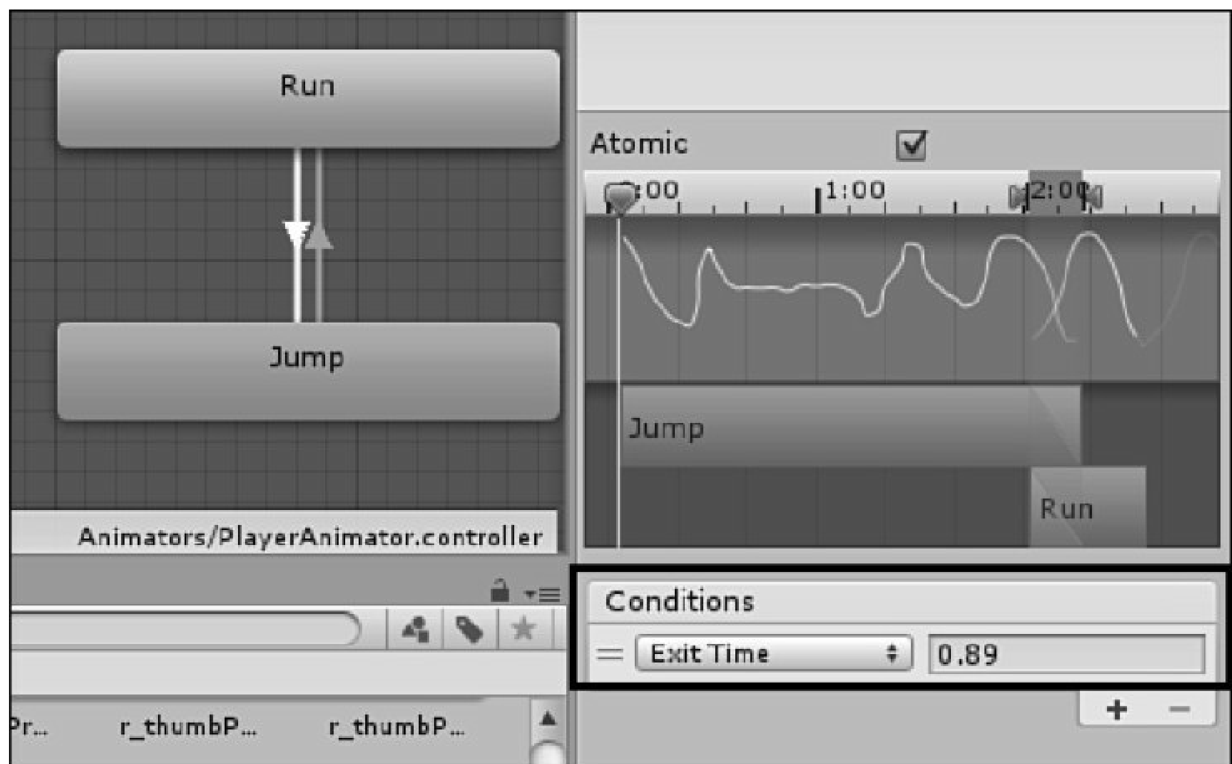


图19.9 Jump过渡属性

既然已经为动画准备了玩家模型，就需要把它放在场景中，如下所示。

（1）找到Robot Kyle.fbx 文件，并把它拖到场景中。然后把机器人定位于(0, .25, −8.5)处。

（2）给模型添加一个胶囊碰撞器（单击Component > Physics > Capsule Collider命令）。然后把碰撞器的y值设置为0.95，并把碰撞器的高度设置为1.72。

（3）把PlayerAnimator拖到Animator组件的Controller属性上，还要确保取消选中Apply Root Motion复选框。

现在应该设置了玩家实体，并使之做好准备。如果运行场景，应该会注意到机器人在奔跑时，赛道在它下面移动。其效果就是机器人看起来好像在向前奔跑一样。

19.4 控制

现在应该添加控制和交互性，以使这款游戏正常运行。由于充电装置和障碍物的位置已经位于预设中，将无需创建一个复活点。因此，大多数控制都将放在一个游戏控制对象上。

19.4.1 触发器区域脚本

你想要创建的第一个脚本是用于触发器区域的脚本。记住：触发器区域将简单地摧毁任何阻挡玩家的对象，以便让玩家通行。要创建它，只需创建一个名为TriggerZoneScript的新脚本，并把它附加到触发器区域游戏对象上。把下面的代码放在脚本中：

```
void OnTriggerEnter(Collider other)
{
  Destroy (other.gameObject);
}
```

触发器脚本非常基本，只会摧毁进入其中的任何对象而已。

19.4.2 游戏控制脚本

大多数工作都是在这个脚本中完成的。首先，在场景中创建一个空的游戏对象，并把它命名为GameControl，它将只是脚本的一个占位符。然后创建一个名为GameControlScript的新脚本，并把它附加到你刚才创建的游戏控制对象上。下面是游戏控制脚本的代码，这里有些复杂，因此一定要仔细阅读每一行，弄明白它正在做什么。把下面的代码添加到脚本中：

```
public float objectSpeed = -.3f;
float minSpeed = -.15f;
float maxSpeed = -.3f;
public GroundScript ground;
public GroundScript wall1;
public GroundScript wall2;
float timeRemaining = 10;
float timeExtension = 1.5f;
float totalTimeElapsed = 0;
bool isGameOver = false;
void Update () {
    if(isGameOver)
        return;
    totalTimeElapsed += Time.deltaTime;
    timeRemaining -= Time.deltaTime;
    if(timeRemaining <= 0)
    isGameOver = true;
}
public void SlowWorldDown()
{
    CancelInvoke("SpeedWorldUp");
    objectSpeed = minSpeed;
    ground.SlowDown();
    wall1.SlowDown();
    wall2.SlowDown();
    Invoke ("SpeedWorldUp", 1);
}
```

```
void SpeedWorldUp()
{
    objectSpeed = maxSpeed;
    ground.SpeedUp();
    wall1.SpeedUp();
    wall2.SpeedUp();
}
public void PowerupCollected()
{
    timeRemaining += timeExtension;
}
void OnGUI()
{
    if(!isGameOver)
    {
        GUI.Box(new Rect(Screen.width / 2 - 50, Screen.height - 100, 100,
50), "Time Remaining");
        GUI.Label(new Rect(Screen.width / 2 - 10, Screen.height - 80, 20,
40), ((int)timeRemaining).ToString());
    }
    else
    {
        GUI.Box(new Rect(Screen.width / 2 - 60, Screen.height / 2 - 100,
120, 50), "Game Over");
        GUI.Label(new Rect(Screen.width / 2 - 55, Screen.height / 2 - 80,
90, 40), "Total Time: " + (int)totalTimeElapsed);
    }
```

}

记住：这款游戏的前提条件之一是当玩家撞上某个障碍物时，所有的一切都会降低速度。因此，对象将需要从游戏控制获得它们的速度。前3个变量分别是对象的当前、最小和最大速度。你还将跟踪地面和墙壁，以便可以根据需要降低它们的速度。其余的变量用于维持游戏的时间设置和状态。

Update()方法用于记录时间，它将把从上一帧起经过的时间（Time.deltaTime ）加到totalTimeElapsed变量上。它还会检查游戏是否结束，当剩余的时间到达0时就会发生。如果游戏结束，它就会设置isGameOver标志。

SlowWorldDown()和 SpeedWorldUp()方法协同工作。无论何时玩家撞上障碍物，都会调用 SlowWorldDown()方法，该方法实质上会减慢场景中的所有对象的速度。它然后会调用Invoke()方法，该方法实质上指示“在x秒内调用这里编写的方法”，其中调用的方法是在引号中指定的方法，秒数则是第二个值。你可能注意到在 SlowWorldDown()方法开头调用了CancelInvoke()方法，这实质上会取消等待调用的任何SpeedWorldUp()方法，因为玩家撞上了另一个障碍物。在上面的代码中，1 秒钟后将会调用 SpeedWorldUp()方法。该方法将恢复一切对象的速度，使得可以像正常的那样继续玩游戏。

PowerupCollected( )方法由玩家调用，用于把延长的时间加到剩余的时间上。

最后，当游戏在运行时，OnGUI方法会把剩余的时间绘制到场景中，一旦游戏结束，还将绘制游戏持续的总时间。

19.4.3 玩家脚本

这个脚本具有两个职责：管理玩家移动和碰撞控制，以及管理动画

器。创建一个名为PlayerScript的新脚本，并把它附加到场景中的机器人模型上。把以下代码添加到脚本中：

```
public GameControlScript control;
Animator anim;
float strafeSpeed = 2;
bool jumping = false;
void Start () {
    anim = GetComponent<Animator>();
}
void Update () {
    transform.Translate(Input.GetAxis("Horizontal") * Time.deltaTime *
    strafeSpeed, 0f, 0f);
    if(transform.position.x > 3)
        transform.position = new Vector3(3, transform.position.y,
transform.
        position.z);
    else if(transform.position.x < -3)
        transform.position = new Vector3(-3, transform.position.y,
        transform. position.z);
    if (anim.GetCurrentAnimatorStateInfo(0).IsName("Base
Layer.Jump"))
    {
        anim.SetBool("Jumping", false);
        jumping = true;
    }
    else
    {
```

```
            jumping = false;
            if(Input.GetButtonDown("Jump"))
                anim.SetBool("Jumping", true);
        }
    }
    void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.name == "Powerup(Clone)")
        {
            control.PowerupCollected();
        }
        else if(other.gameObject.name =="Obstacle(Clone)"&& jumping ==
  false)
        {
            control.SlowWorldDown();
        }
        Destroy(other.gameObject);
    }
```

前两个变量保持游戏控制和动画器引用，接下来两个变量包含与移动相关的信息。anim变量的值是在Start()方法中设置的。

Update()方法首先基于输入移动玩家。然后它将执行检查，以确保玩家在x轴上没有超过−3或3。如果玩家超过了，就把他设置回−3或3，这就把玩家保留在赛道中。Update()方法然后将检查玩家当前是否处于跳跃动画中。如果是，就把局部跳跃标志设置为 true（使得玩家不会与障碍物发生碰撞），并把动画器跳跃参数设置为false（使得跳跃动画不会循环）。如果玩家当前不是在跳跃，动画器就会设置合适的标志，并且检查玩家是否按下了Jump按钮（默认是空格键）。

在OnTriggerEnter()方法中，脚本将检查玩家与什么发生碰撞。如果玩家与充电装置发生碰撞，就会调用合适的方法。为了与障碍物发生碰撞，玩家还绝对不能跳跃。如果是这样，就会调用SlowWorldDown()方法。

19.4.4 充电装置和障碍物的脚本

充电装置和障碍物脚本完全相同。事实上，可以把它们创建成单独一个脚本。这里将把它们保持独立，以便将来可以轻松地执行不同的修改。创建两个脚本，分别命名为PowerupScript和ObstacleScript。把充电装置脚本添加给充电装置预设，其方法是选择预设，并在Inspector中单击Add Component > Scripts > Powerup Script 命令。对障碍物预设和障碍物脚本做相同的事情，然后把以下代码添加到每个脚本中：

```
public GameControlScript control;
void Update (){
   transform.Translate(0, 0, control.objectSpeed);
}
```

这个脚本很简单，有一个用于游戏控制脚本的占位符。然后，在每个 Update()方法调用中，以控制的当前速度移动对象。这样，控制就可以更改场景中的每个对象的速度。

19.4.5 复活脚本

复活脚本负责创建这个场景中的对象。由于位置数据存放在预设中，将把这个脚本放在游戏控制对象上。创建一个名为SpawnScript的新脚本，并把它附加到GameControl对象上。然后把以下代码添加到脚本中：

```
GameControlScript control;
```

```
public GameObject obstacle;
public GameObject powerup;
float timeElapsed = 0;
float spawnCycle = .5f;
bool spawnPowerup = true;
void Start () {
    control = GetComponent<GameControlScript>();
}
void Update () {
    timeElapsed += Time.deltaTime;
    if(timeElapsed > spawnCycle)
    {
        GameObject temp;
        if(spawnPowerup)
        {
            temp = (GameObject)Instantiate(powerup);
            temp.GetComponent<PowerupScript>().control = control;
            Vector3 pos = temp.transform.position;
            temp.transform.position = new Vector3(Random.Range(-3, 4),
            pos.y, pos.z);
            }
            else
            {
                temp = (GameObject)Instantiate(obstacle);
                temp.GetComponent<ObstacleScript>().control = control;
                Vector3 pos = temp.transform.position;
                temp.transform.position = new Vector3(Random.Range(-3, 4),
```

```
                pos.y, pos.z);
            }
            timeElapsed -= spawnCycle;
            spawnPowerup = !spawnPowerup;
        }
    }
```

这个脚本从一个指向游戏控制脚本的引用开始。它还包含一个指向充电装置和障碍物游戏对象的引用。接下来的变量用于控制对象复活的时间设置和顺序。充电对象和障碍物将轮流复活，因此有一个标志用于跟踪哪个对象在复活过程中。

在Update()方法中，增加流逝的时间，然后检查此时是否应该复活一个新对象。如果是，脚本然后将检查它应该复活哪个对象。然后，它将复活充电装置或障碍物，并把指向游戏控制脚本的引用传递到新对象的脚本中。这样，充电装置和障碍物就知道在哪里查找游戏控制脚本。然后，随机地左移或右移创建的对象。最后，Update()方法将减少流逝的时间，并且翻充电装置的标志，使得下一次将复活相反的对象。

19.4.6 把游戏的各个部分结合起来

这是游戏的最后一部分，需要把脚本和对象链接在一起。首先在Hierarchy 视图中选择GameControl对象，并把Ground 对象和两个Wall 对象拖到Game Control Script 组件中它们对应的属性上，如图19.10 所示。然后把Powerup 和Obstacle 预设拖到Spawn Script 组件中它们对应的属性上。

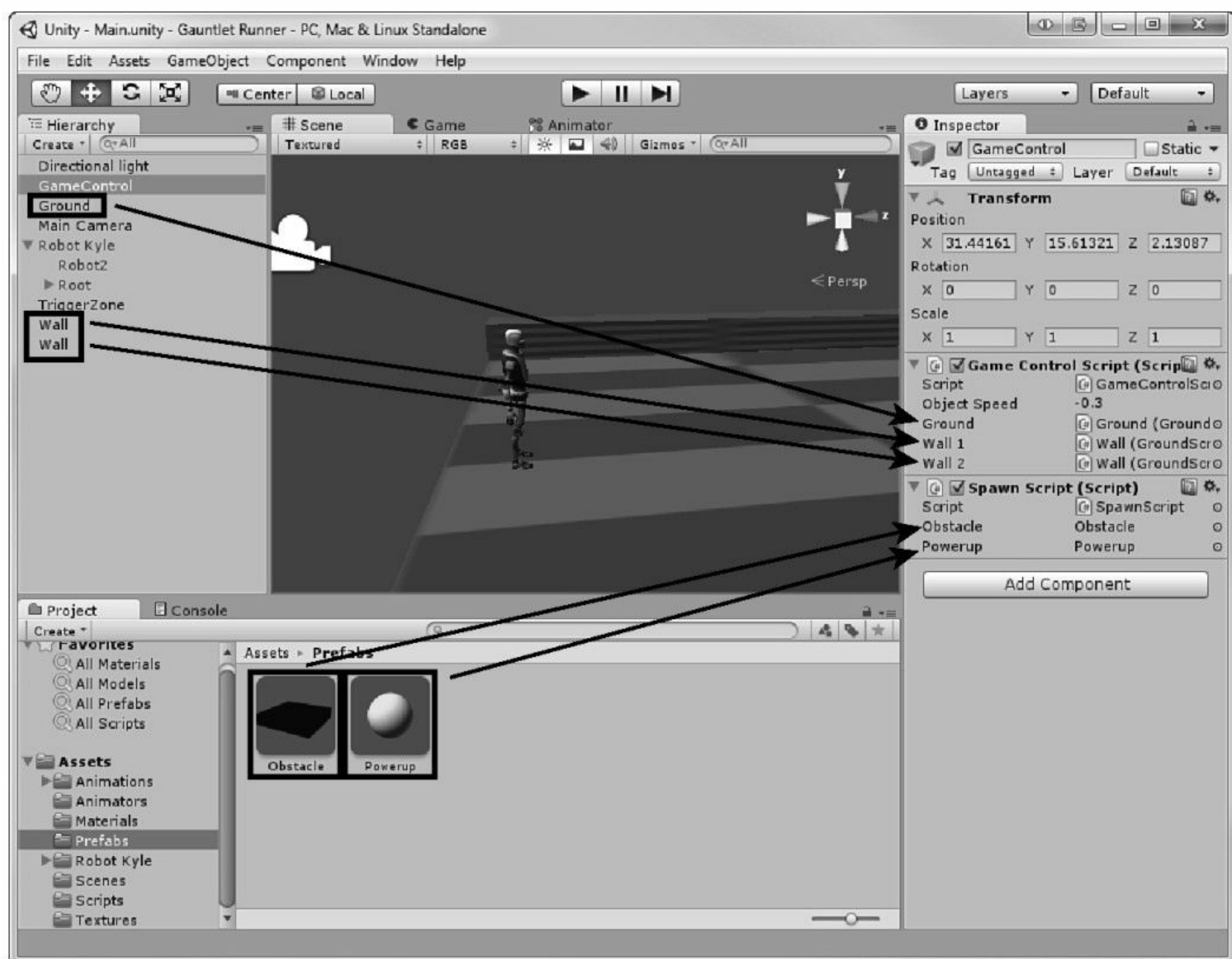


图19.10 把对象拖到它们的属性上

接下来，在 Hierarchy 视图中选择 Robot Kyle 模型，并把GameControl 对象拖到 Player Script组件的Control属性上，如图19.11所示。

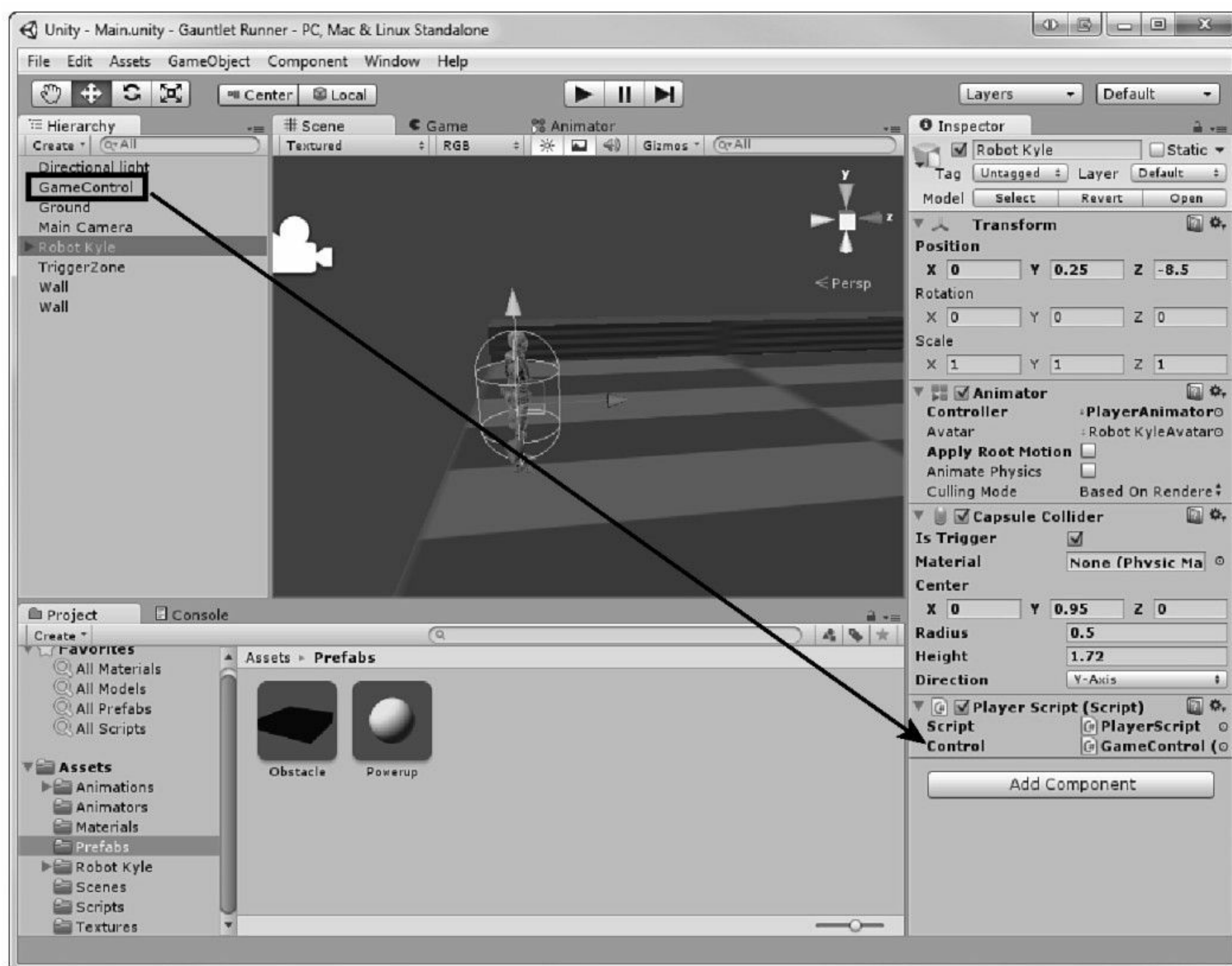


图19.11 把游戏控制添加给玩家脚本

这就行了！游戏现在就完成了，可以开始玩游戏了。

19.5 改进的空间

与以往一样，直到对游戏进行了测试和调整之后，游戏才算是彻底完成了。现在，你应该从头至尾玩一遍游戏，看看你喜欢什么以及不喜欢什么。记住：要把你认为确实可以增强游戏体验的特性记录下来，还要把你感觉有损游戏体验的项目也记录下来。对于有关游戏的将来迭代的任何想法，一定要给出注释说明。尝试让一些朋友也来玩游戏，并且记录下他们关于游戏的反馈。所有这些都有助于使游戏变得独特并且更令人愉悦。

19.6 小结

在本章中，你制作了Gauntlet Runner游戏。你首先布置了游戏的设计元素。接着，你构建了赛道，并且使用纹理技巧使之滚动。然后，你为游戏构建了多个实体。接着，你构建了多个控制和脚本。最后（并非最不重要的），你测试了游戏并且记录了一些反馈。

19.7 问与答

问：对象和地面的移动并不是完全对齐的，这正常吗？

答：在这种情况下，这是正常的。需要进行更精细的测试和调整，以使它们完美地同步。这是一个你可以集中精力进行改进的元素。

问：跳跃动画看上去有一点偏，这正常吗？

答：同样，在这种情况下是正常的。本章中使用的动画是由Unity为它们的Mecanim演示提供的。因此，将不会精确地以设计它们的方式来使用它们（它们打算用于控制玩家的移动）。因此，它看上去有一点偏。有时，游戏开发就是利用所提供的工具做自己可以做的事情。

19.8 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

19.8.1 问题

1．玩家怎样会输掉游戏？

2．滚动式背景如何工作？

3．你在动画器中创建了哪两种状态？

4．游戏怎样控制场景中的所有对象的速度？

19.8.2 答案

1．当时间用完时，就会输掉游戏。

2．赛道将保持静止不动。纹理不会移动，而是沿着对象滚动。结果就是地面看上去好像在移动。

3．Run和Jump。

4．场景中的每个对象都具有一个指向游戏控制脚本的引用，这个脚本自身具有对象应该行进的速度。每次对象更新时，它都会从控制脚本获取速度，看看它应该行进得多快。

19.8.3 练习

你现在应该尝试实现在测试这款游戏时记录的一些修改。你应该尝试使游戏对你来说是独特的。顺利的话，你将能够确定想要改进的游戏的一些弱点或者一些长处。下面列出了一些你可能会考虑更改的方面。

尝试添加新的/不同的充电装置和障碍物。

尝试改进对象速度，以便更好地对齐滚动的地面。

尝试通过更改充电装置和障碍物复活的频率，来增加或减小难度。还可以更改充电装置增加了多少时间或者游戏世界变慢了多长时间。甚至可以尝试调整游戏世界变慢的程度，或者给不同的对象提供不同的变慢速度。

给充电装置和障碍物提供一种新的外观。试验一些纹理和粒子效果，使它们看上去非常棒。

第20章 音频

在本章中你将学到：

Unity中的音频的基本知识；

怎样使用音频源；

怎样通过脚本处理音频。

在本章中，你将学习 Unity 中的音频，首先将了解音频的基本知识。接着，将探索音频源组件以及它们是如何工作的。在这个过程中，你还将查看单独的音频剪辑及其作用。最后，你将学习如何利用代码操纵音频。

20.1 音频的基本知识

任何体验都与它自身的声音息息相关。比如一部恐怖电影，给它添加一种笑声配乐，一下子便会让紧张的体验就变成了有趣的体验。视频游戏也是如此。大多数时间玩家并没有认识到这一点，但是声音在整个游戏中占了非常大的比重。当玩家在解密时，声音将给出暗示，比如铃声标记。咆哮的加农炮可以给战争模拟游戏增加一点现实感。使用Unity，很容易实现令人惊异的音频效果。

20.1.1 音频的组成部分

为了使音频在场景中工作，需要3个组件：音频侦听器、音频源和音频剪辑。音频侦听器是音频系统中最基本的组件，侦听器是一个简单的组件，其唯一职责是“倾听”场景中发生的事情。为了更容易理解，你可以把它们视为就像你的世界里的耳朵一样。默认情况下，每个场景都开始于附加到Main Camera 上的音频侦听器，如图20.1 所示。没有可供音频侦听器使用的属性，并且不需要做任何事情以使之工作。把音频侦听器放在代表玩家的任何游戏对象上是一种常见的实践。注意：如果把音频侦听器放在任何其他的游戏对象上，则将需要从Main Camera 上移除它。每个场景中只允许有一个音频侦听器。

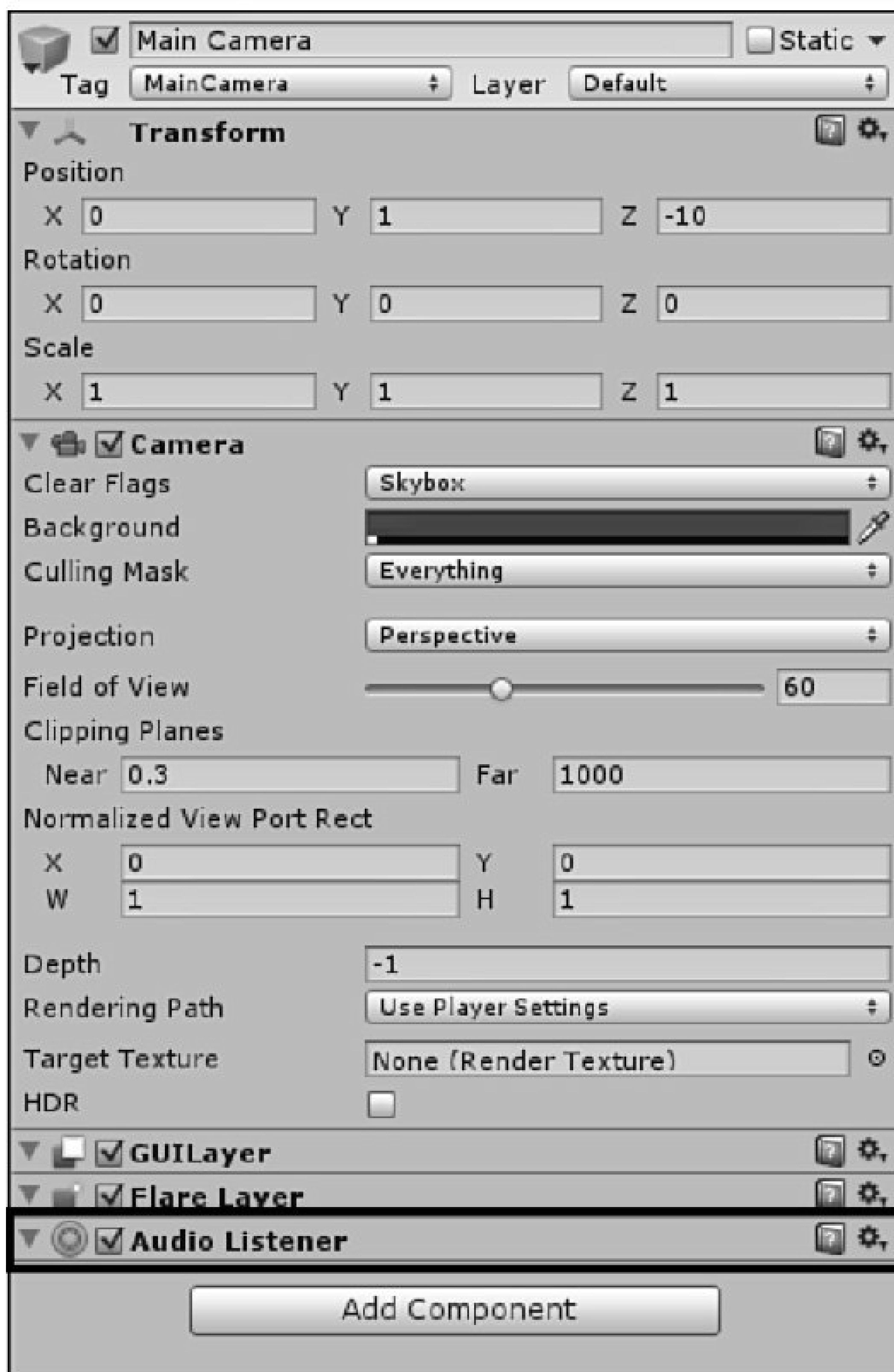

Main Camera
Static
Tag
MainCamera
Layer
Default
Transform
Position
X 0
Y 1
Z -10
Rotation
X 0
Y 0
Z 0
Scale
X 1
Y 1
Z 1
Camera
Clear Flags
Skybox
Background
Culling Mask
Everything
Projection
Perspective
Field of View
60
Clipping Planes
Near 0.3
Far 1000
Normalized View Port Rect
X 0
Y 0
W 1
H 1
Depth
-1
Rendering Path
Use Player Settings
Target Texture
None (Render Texture)
HDR
GUILayer
Flare Layer
Audio Listener
Add Component

图20.1 音频侦听器

音频侦听器用于侦听声音，但是实际发出声音的是音频源，这是一个可以放在场景中的任何对象（甚至是它上面带有音频侦听器的对象）上的组件。有许多属性和设置涉及音频源，在本章后面将用专门的小节介绍它们。

正常工作的音频所需的最后一个项目是音频剪辑。正如你所想，音频剪辑是由音频源实际地播放的声音文件。每个剪辑都具有一些属性，可以设置它们来改变 Unity 播放它们的方式。Unity支持以下音频格式：.aif、.wav、.mp3和.ogg。这3个项目一起给场景提供了音频体验。

20.1.2 2D和3D音频

关于音频，需要知道的一个概念是2D和3D音频的思想。2D音频剪辑是最基本的音频类型，无论音频侦听器是否接近于场景中的音频源，它们都会以相同的（最大）音量播放。2D 声音最适用于菜单、警告、声道或者总是必须以完全相同的方式被收听的任何音频。2D音频最大的优点同时也是它们最大的弱点。考虑一下，无论你身处何方，游戏中的每种声音都以完全相同的音量播放，它将很快失去控制。

3D音频解决了2D音频的问题。这些音频剪辑具有称为衰减（roll off）的特点，它规定依赖于音频侦听器距离音频源有多近，声音将怎样变得更小或更大。在高级音频系统（比如Unity的音频系统）中，3D声音甚至可以具有一种模拟的多普勒效应（后面将更详细地介绍它）。如果在充斥有不同音频源的场景中寻找逼真的音频，3D音频就是你要努力的目标。

不同音频剪辑的维度是在声音文件的各个设置中管理的。

20.2 音频源

如前所述，音频源是场景中实际地播放音频剪辑的组件。这些音频源与音频侦听器之间的距离确定了3D音频剪辑如何发声。要给游戏对象添加音频源，可以选择想要的对象，并单击Component > Audio > Audio Source命令。

音频源组件具有一系列属性，可以让你精细地控制在场景中如何播放声音。表20.1描述了音频源组件的多个属性。

表20.1 音频源的属性

属性	描述
Audio Clip	要播放的实际声音文件
Mute	确定是否进行静音
Bypass Effects	确定是否将音频效果（仅 Pro 版本）应用于这种音频源。选择这个属性将关闭这些效果
Bypass Listener Effects	确定是否将音频侦听器效果（仅 Pro 版本）应用于这种音频源。选择这个属性将关闭这些效果
Bypass Reverb Zone	确定是否将混响区域效果（仅 Pro 版本）应用于这种音频源。选择这个属性将关闭这些效果
Play On Wake	确定是否一创建了音频源就开始播放它
Loop	确定是否音频源一播放完就重新开始音频剪辑
Priority	音频源的重要性。0 是最重要的，255 是最不重要的。背景音乐应该设置为，以避免被换出
Volume	音频源的音量，其中 1 等价于 100%的音量
Pitch	音频源的音高标准
3D Sound Settings	适用于 3D 音频剪辑的设置，将在后面更详细地介绍
2D Sound Settings	适用于 2D 音频剪辑的设置，将在后面更详细地介绍

注意：

音频的优先级

每个系统都具有数量有限的音频声道。这个数量并不是一致的，它依赖于许多因素，比如系统的硬件和操作系统。因此，大多数音频系统

都会利用一种优先级系统。在优先级系统中，声音是以如下顺序播放的：直到使用了最大数量的声道，才会接收到它们。一旦使用了所有的声道，就会为高优先级的声音换出低优先级的声音。只需记住：在Unity中，优先级数字越低，意味着实际的优先级越高！

20.2.1 导入音频剪辑

如果没有任何音频要播放，那么音频源将无所事事。在 Unity 中，导入音频就像导入其他任何内容一样容易。只需单击想要的文件并将其拖到Project视图中，即可把它们添加到资源中。Jeremy Handel 大度地把这些音频文件提供给你使用（http://handelabra.com/）。

测试音频

让我们在 Unity 中测试音频，并且确保一切都会工作。一定要保存这个场景，因为将在下一节中使用它。

（1）创建一个新的项目或场景。然后在用于第20章（Hour 20）的本书配套资源中找到Sounds文件夹，并把它拖到Unity中Project视图下的Assets文件夹中。

（2）在场景中创建一个立方体，并把它定位在(0, 0, 0)处。给这个立方体添加一个音频源（单击 Component > Audio > Audio Source 命令）。在新导入的 Sounds 文件夹中找到looper.ogg 文件，并把它拖到立方体上的音频源的Audio Clip 属性中，如图20.2所示。

（3）确保选中Play On Awake 属性，并运行场景。注意声音在播放。音频应该会在大约20秒后停止（除非把它设置为循环播放）。

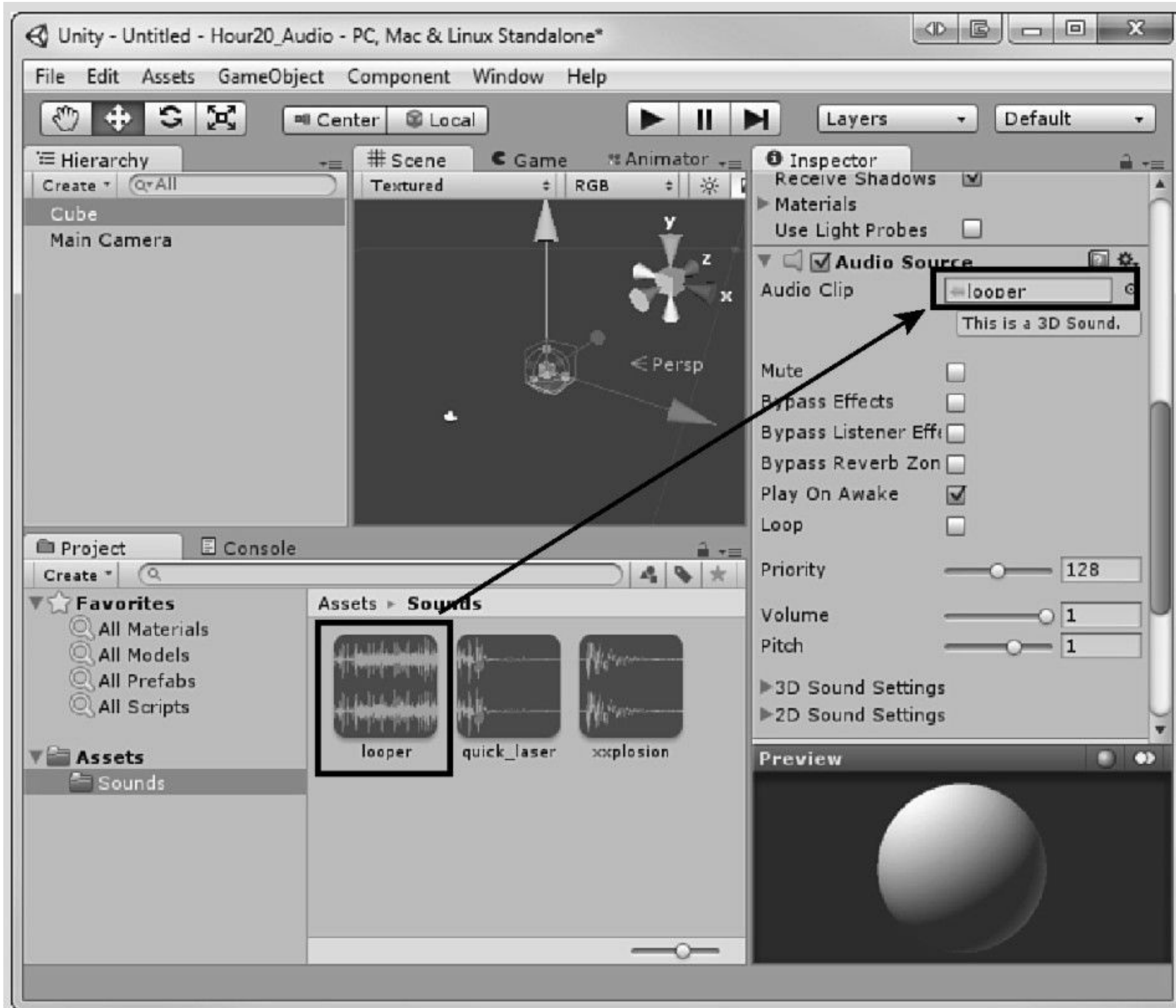


图20.2 给音频源添加一个剪辑

20.2.2 在Scene视图中测试音频

如果每次想要测试音频时都需要运行场景，那就有点劳神了。这不仅需要启动场景，还需要导航到游戏世界中的场景。这并不总是容易（或者甚至可能）做到。作为替代，可以在Scene视图中测试音频。

要在Scene视图中测试音频，需要打开场景音频，其方法是单击场景音频切换开关，如图20.3所示。在这样做时，将使用假想的音频侦听器。这个侦听器位于Scene视图中的引用的帧上（而不是位于实际的音

频侦听器组件所在的位置）。

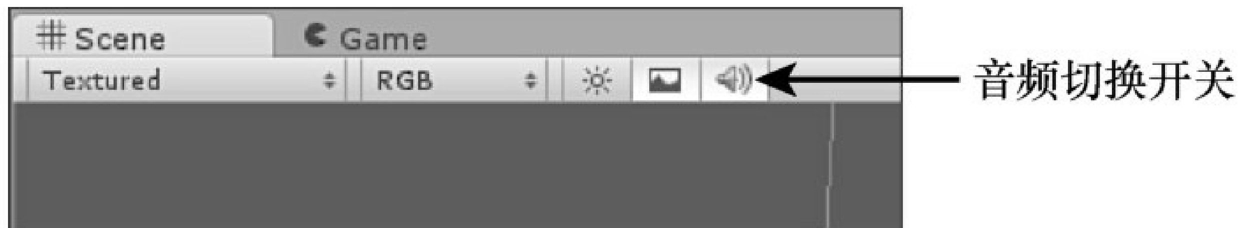


图20.3 音频切换开关

Scene视图中的音频

这个练习将显示如何在Scene视图中测试音频，它将使用在前一个练习中创建的场景。

（1）打开或创建前一个练习中的场景。

（2）打开场景音频切换开关，如图20.3所示。

（3）四处移动Scene视图。注意声音怎样基于相对于发出声音的立方体的距离而变大和变小。默认情况下，所有的声音剪辑都是3D，因此会受到源与侦听器之间的距离支配。

20.2.3 3D音频

如前所述，所有的音频默认都设置为 3D。这意味着所有的音频都将受到基于距离和移动的 3D 音频效果限制，可以通过音频组件的 3D 属性修改这些效果，如图 20.4 所示。

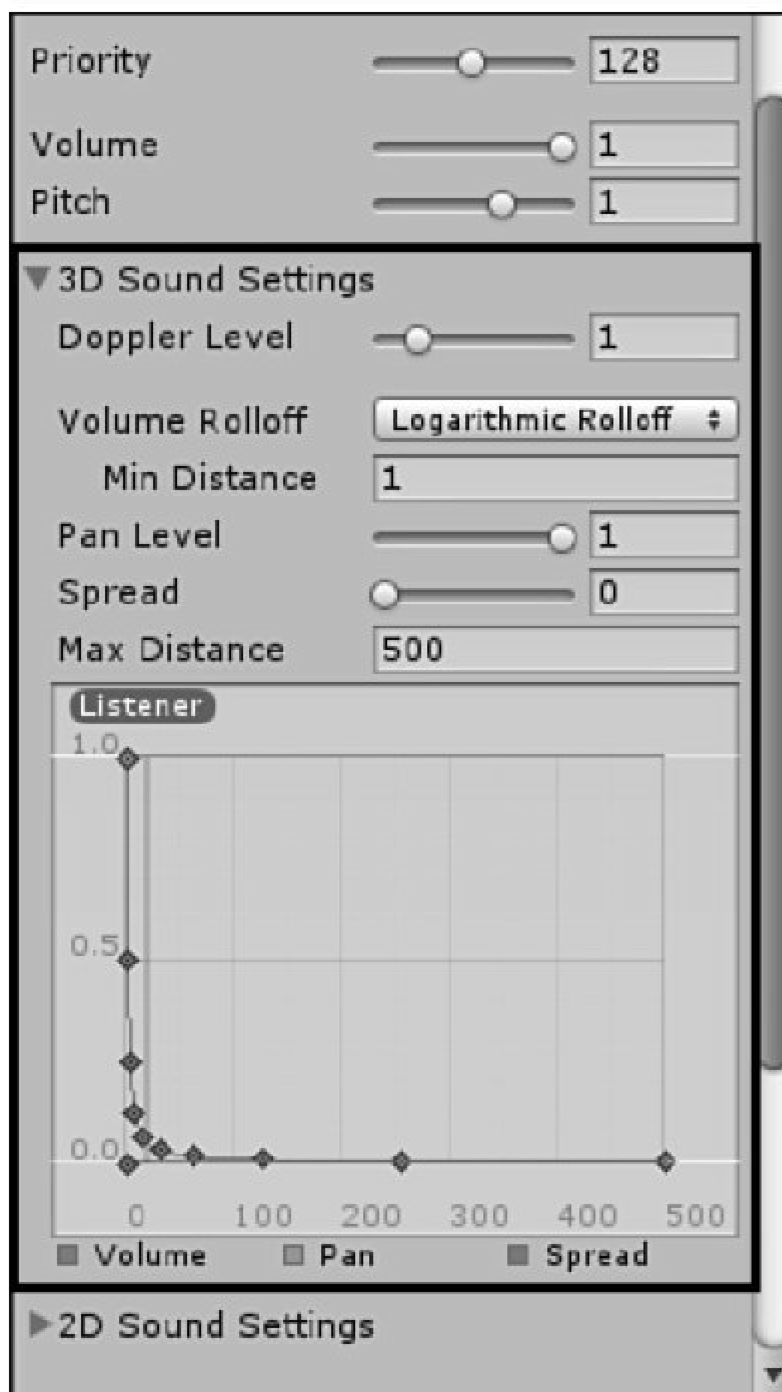


图20.4 3D音频设置

表20.2描述了多个3D音频属性。

表20.2 3D音频属性

属性	描述
Doppler Level	确定对音频应用多少多普勒效应，设置 0 意味着将不会应用任何效应。多普勒效应是指在朝着声音走去或者离开它时声音是如何失真的
Volume Rolloff	确定音量如何随着距离的改变而改变，默认设置为 Logarithmic。也可以选择 Linear，或者利用自定义的衰减设置你自己的曲线
Min Distance	在接收到 100%的音量之前必须离开音频源的距离。这个数字越高，就可以离开得越远，并且仍然可以接收到 100%的音量
Pan Level	确定 3D 引擎对音频具有多大的影响。把它设置为 0 就相当于具有一种 2D 声音
Spread	系统的多个扬声器是怎样分布的。把它设置为 0 就意味着所有的扬声器都处于相同的位置，并且信号实质上是一种单声道信号。可以不理会这个属性，除非你更多地理解了音频系统
Max Distance	离开音频源并且仍然可以听到某个音量的最远距离

20.2.4 2D音频

有时，无论音频处于场景中的什么位置，你都希望它以最大音量播放。它的一个最常见的示例是背景音乐。要把音频剪辑从3D切换为2D，可以选择音频文件，并在Inspector视图中取消选中3D Sound 属性，如图20.5所示。

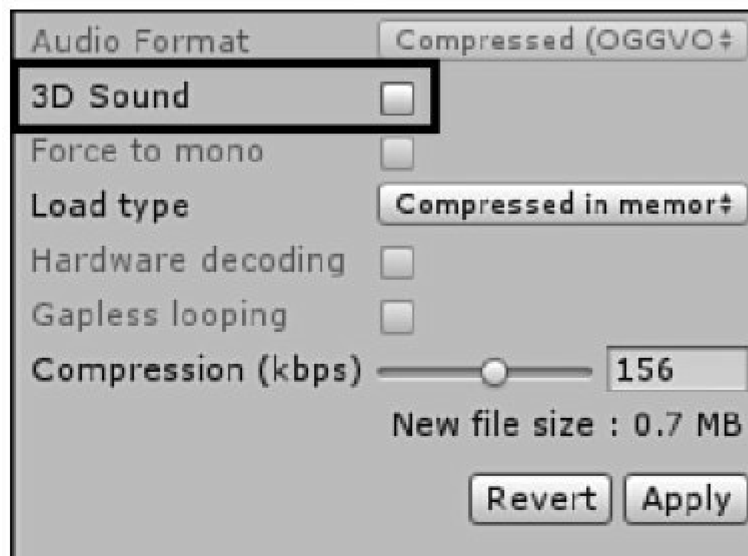


图20.5 把声音剪辑设置为2D

一旦把音频剪辑设置为2D，就只会给它应用音频源的2D Sound Settings 中的属性。非常简单，只有一个2D属性：Pan。这个实例中的Pan属性用于控制如何播放音频。如果把它的值设置为 0，声音将在立

体声环境中从两个扬声器（左边和右边）同样地播放出来。如果把它的值设置为−1，音频将只会在左边播放；如果把值设置为1，则会使音频只在右边播放。

测试2D音频

让我们在一个场景中试验2D音频。这个练习将使用在前两个练习中创建的场景。

（1）打开或创建前面练习中的场景。

（2）在Sounds文件夹中，找到looper.ogg文件并选择它。然后在Inspector视图中，取消选中3D Sound 属性，并单击Apply按钮。

（3）打开场景音频切换开关，并在场景中四处导航。注意立方体的位置对播放的声音没有影响。

20.3 编写音频的脚本

在创建音频源时播放音频将非常有用，假定这就是你所想要的功能。不过，如果你希望等待并在某个时间播放声音，或者从相同的音频源播放不同的声音，将需要使用脚本。幸运的是，通过代码管理音频不是太困难。它的大部分工作就像你已经习惯的任何音频播放器一样。只需选择一首歌曲并按下Play即可。所有的音频脚本都是使用变量和方法完成的，它们是对象音频的一部分。

20.3.1 启动和停止音频

你可能想要的最基本的功能只是简单地启动和停止音频剪辑，这些是由两个简单的名为Start( )和Stop()的方法控制的。可以像下面这样使用这些方法：

```
audio.Start(); //Starts a clip
audio.Stop(); //Stops a clip
```

这段代码将会播放通过音频源组件的Audio Clip 属性指定的剪辑。你还能够在一段延迟之后启动剪辑，为此，将使用PlayDelayed( )方法，该方法接受单个参数，它是在播放剪辑前等待的时间（以秒为单位）。这个方法如下所示：

```
audio.PlayDelayed(<some time in seconds>);
```

可以通过在代码中检查 isPlaying 变量（它是音频对象的一部分），辨别某个剪辑当前是否正在播放。要访问这个变量，从而弄明白剪辑是否正在播放，可以输入以下代码：

```
if(audio.isPlaying)
```

```
{
   //The track is playing
}
```

顾名思义，如果音频当前正在播放，这个变量就为“真”；否则，它将为“假”。

启动和停止音频

在这个练习中，将使用脚本启动和停止一个音频剪辑。

（1）创建一个新的项目或场景。如果还没有导入Sounds文件夹，就要从本书配套资源中导入它。在场景中放入一个立方体，并把它定位于(0, 0, 0)处，然后把一个音频源放在它上面。

（2）从Sounds文件夹中把looper.ogg文件拖到立方体上的音频源的Audio Clip属性上，还要确保取消选中音频源的Play On Wake 属性，但要选中Loop 属性。

（3）创建一个名为Scripts的新文件夹，并在其中创建一个名为AudioScript的新脚本，把该脚本附加到立方体上。然后向脚本中添加以下代码：

```
void Update () {
   if(Input.GetButtonDown("Jump"))
   {
      if(audio.isPlaying == true)
      {
         audio.Stop();
      }
      else
      {
         audio.Play();
      }
```

```
    }
}
```

（4）播放场景，可以按下空格键启动和停止音频。注意每次播放音频时音频剪辑是怎样重新启动的。

提示：

未提到的属性

Inspector视图中列出的音频源的所有属性也可通过脚本使用。例如， Loop属性在代码中是利用audio.loop变量访问的。如前所述，所有这些变量都是用于音频对象的，看看你可以找到多少个这样的变量。

20.3.2 更改视频剪辑

通过脚本可以轻松地控制要播放哪些音频剪辑。关键是在使用Play( )方法播放剪辑之前，在代码中更改音频剪辑属性。在切换到一个新的音频剪辑之前，总是要确保停止当前的音频剪辑；否则，剪辑将不会切换。

要更改音频源的音频剪辑，可以给对象audio的clip变量分配一个AudioClip类型的变量。例如，如果具有一个名为newClip的音频剪辑，可以使用以下代码把它分配给音频源，并播放它：

```
audio.clip = newClip;
audio.Play();
```

可以轻松地创建一个音频剪辑的集合，并以这种方式切换它们。

20.4 小结

在本章中，你学习了如何在 Unity 中使用音频。首先学习的是音频的基本知识，以及使之工作所需的组件。接着，你探索了音频源组件，学习了如何在Scene视图中测试音频，以及如何使用2D和3D音频剪辑。在本章最后，你学习了通过脚本操纵音频。

20.5 问与答

问：一个系统平均具有多少个音频声道？

答：这因系统而异。不过，知道耳机一般只具有两个音频声道是一种良好的基本知识。这并不意味着耳机只能播放两种声音，它只意味着在系统需要把声音混合在一起之前只能有两种声音在播放，混合声音会使质量稍稍降级。

20.6 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

20.6.1 问题

1．工作的视频需要什么项目？

2．判断题：无论侦听器与音频源的距离有多远，3D声音都会以相同的音量播放？

3．哪个方法允许在一段延迟之后播放音频剪辑？

20.6.2 答案

1．音频侦听器、音频源和音频剪辑。

2．错误。2D声音以相同的音量播放。

3．PlayDelayed()方法。

20.6.3 练习

在这个练习中，你将创建一个基本的音板。这个音板将允许播放3种声音之一。你还能够启动和停止声音，以及启用和禁止循环。在用于第20章（Hour 20）的本书配套资源中可以找到完成的练习，其名称为Hour20_Exercise。

1．创建一个新的项目或场景。向场景中添加一个立方体，并把它定位于(0, 0,−10) 处，然后给立方体添加一个音频源。一定要取消选中Play On Wake 属性。在用于第20章（Hour 20）的本书配套资源中找到Sounds 文件夹，并把它拖到Assets 文件夹中。

2．创建一个名为Scripts的新文件夹，并在其中创建一个名为AudioScript的新脚本，把该脚本附加到立方体上。然后更改脚本，使之包含以下代码：

```
public AudioClip clip1;
public AudioClip clip2;
public AudioClip clip3;
void Start()
{
  audio.clip = clip1;
}
void Update () {
  if(Input.GetButtonDown("Jump"))
  {
    if(audio.isPlaying)
      audio.Stop();
    else
      audio.Play();
    }
    if(Input.GetKeyDown(KeyCode.L))
      audio.loop = !audio.loop; //toggles lopping
    if(Input.GetKeyDown(KeyCode.Alpha1))
    {
      audio.Stop();
      audio.clip = clip1;
      audio.Play();
    }
    else if(Input.GetKeyDown(KeyCode.Alpha2))
```

```
        {
            audio.Stop();
            audio.clip = clip2;
            audio.Play();
        }
        else if(Input.GetKeyDown(KeyCode.Alpha3))
        {
            audio.Stop();
            audio.clip = clip3;
            audio.Play();
        }
    }
```

3．在 Unity 编辑器中，选取场景中的立方体。从 Sounds 文件夹中把 looper.ogg、quick_laser.ogg和xxplosion.off这些音频文件分别拖到音频脚本的Clip1、Clip2和Clip3属性上。

4．运行场景，注意你可以怎样利用数字键1～3更改音频剪辑，也可以利用空格键启动和停止音频。最后，你可以利用L键切换循环方式。

第21章 移动开发

在本章中你将学到：

怎样为移动开发做准备；

怎样使用设备加速计；

怎样使用设备触摸屏。

诸如手机和平板电脑之类的移动设备正变成常见的游戏设备。在本章中，你将学习利用Unity为Android和iOS设备进行移动开发，首先将探讨移动设备的需求。接着，你将学习如何接受来自设备加速计的特殊输入。最后，你将学习触摸界面输入。

注意：

需求

本章专门介绍了移动设备开发。因此，如果你没有移动设备（iOS或Android），将不能遵照任何动手练习来操作。不过，也不要担心，阅读材料仍然应该是有意义的，并且你仍然能够在移动设备上制作游戏，而只是不能玩这些游戏。

21.1 为移动做准备

Unity使得为移动设备开发游戏非常容易。从Unity版本4.1起，移动插件甚至是免费的！你还将很高兴地知道为移动平台开发游戏与为其他平台开发游戏几乎完全相同。这意味着只需构建一次游戏，就可以把它部署在任何系统上。不再有任何问题阻止你为所有主要的平台构建游戏。这种跨平台的兼容性级别是前所未有的。不过，在你开始在 Unity 中处理移动设备之前，需要把计算机设置和配置成能够这样做。

注意：

大量的设备

有许多不同类型的移动设备。在编写本书时，Apple具有3种设备（iPod、iPad 和 iPhone），Android 则具有许多种类的手机和平板电脑。其中每种设备都具有稍微不同的硬件以及正确配置它们的步骤。因此，本书只是简单地尝试指导你通过安装过程，而编写适合于每一个人的准确指南是不可能的。事实上，Unity、Apple和Google已经编写了多份指南，它们比本书更好地解释了这个过程，需要时可以参考它们。

21.1.1 设置环境

甚至在打开 Unity 制作游戏之前，就需要设置开发环境。它的特定细节因目标设备和你尝试做什么而异，但是一般的步骤如下。

（1）安装目标设备的软件开发工具包（software development kit，SDK）。

（2）确保计算机识别并且可以处理你的设备（仅当你想在设备上执行测试时它才是重要的）。

（3）告诉Unity在哪里查找SDK（仅Android需要）。

如果这些步骤对你而言似乎有点神秘，也不要担心。有大量的资源可用于帮助你执行这些步骤，最好的起步是利用 Unity 自己的文档。可以在 http://docs.unity3d.com 上访问 Unity的文档。

这个站点包含关于 Unity 的方方面面的实时文档。默认情况下，它只会显示与桌面开发相关的项目。你需要启用针对Android和iOS的文档，应该会看到Android和iOS图标带有一个红色的“×”，如图21.1所示。

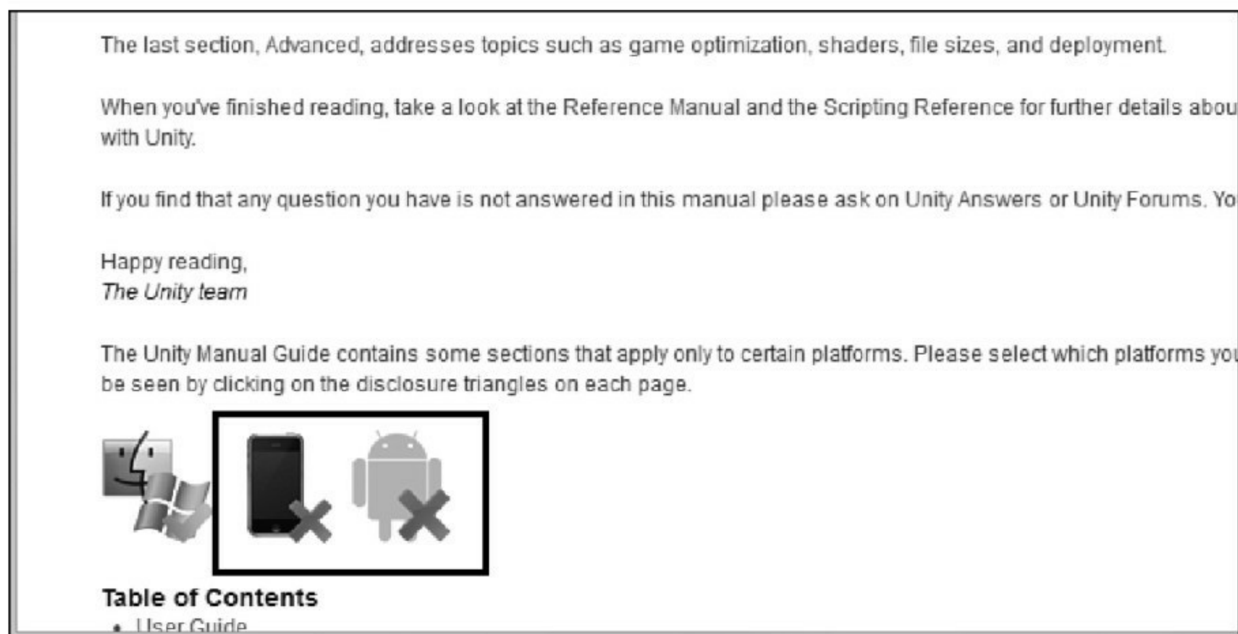


图21.1 禁用的移动设备图标

单击其中任何一个项目都将在它们上面显示一个绿色的“√”，并启用该文档，如图21.2所示。在图21.2中可以看到，Unity文档具有一些指导，可以帮助你设置iOS和Android 环境。当设置环境的步骤改变时，这些文档也会随之更新。在完成了为目标环境配置开发环境的步骤之后，或者如果你没有计划详细介绍某个设备，就可以继续学习下一节的内容。

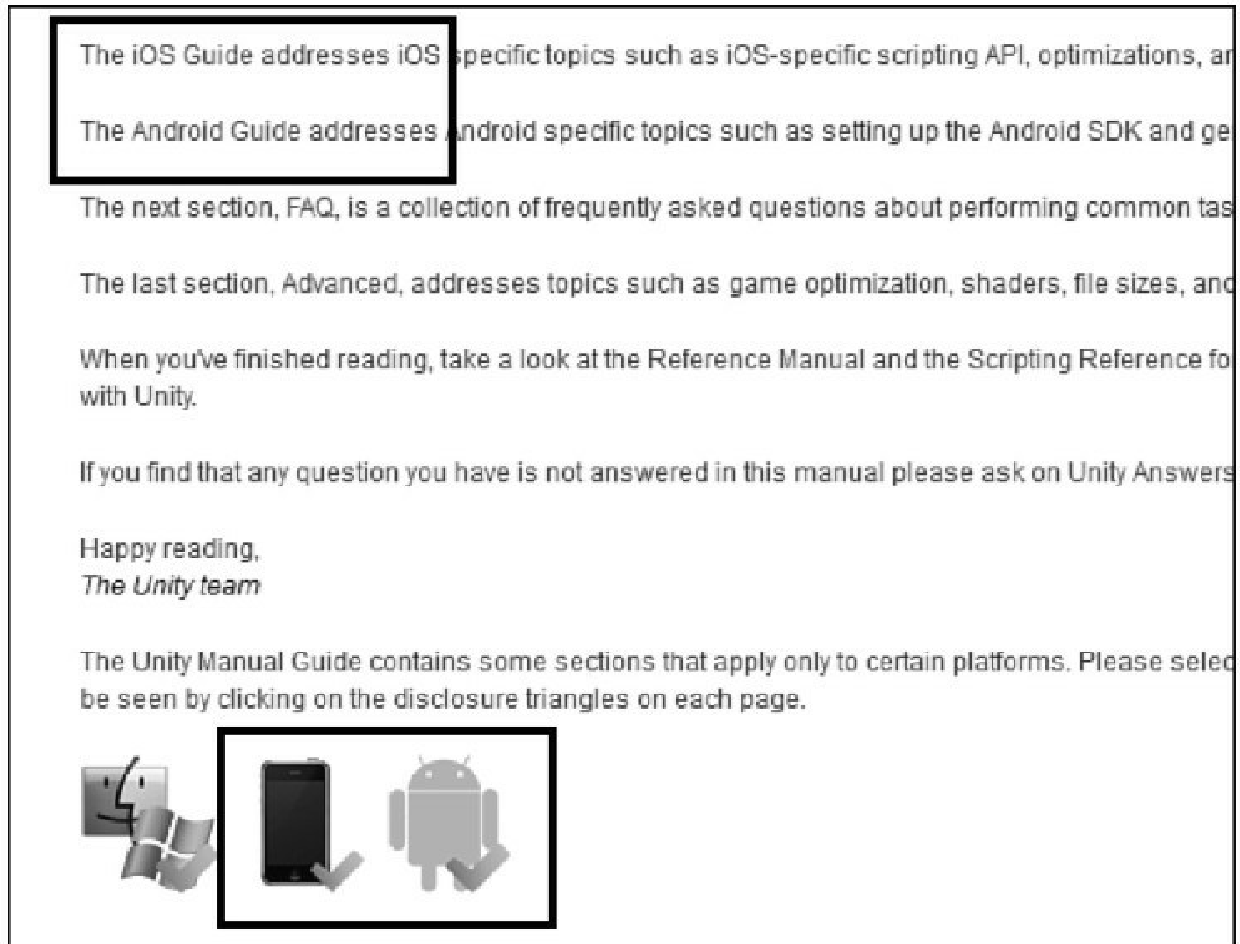


图21.2 启用移动文档

21.1.2 Unity Remote

在设备上测试游戏的最基本的方式是构建项目，把得到的文件放到设备上，然后运行它。这可能是一个笨拙的系统，你肯定很快就会感到疲劳。测试游戏的另一种方式是构建项目，然后通过iOS或Android仿真器运行它。同样，这需要相当多的步骤，并且涉及配置和运行仿真器。如果你是在对高级特性（比如性能和渲染）执行广泛的测试，这些系统就可能是有用的。不过，对于基本的测试，有一种要好得多的方式，即Unity Remote。

Unity Remote 是一个应用程序，可以从移动设备的应用程序商店中

下载它，用于测试移动设备上的应用程序，同时仍然在 Unity 编辑器中运行它。简而言之，这意味着在开发的同时可以实时体验在设备上运行的游戏，并且使用设备把设备输入发回给游戏。在http://docs.unity3d.com/Documentation/Manual/unity-remote.html 上可以找到关于Unity Remote的更多信息。

要找到Unity Remote应用程序，可以在设备的应用程序商店中搜索名词“Unity Remote”。从此处，可以像其他任何应用程序一样下载并安装它，如图21.3所示。

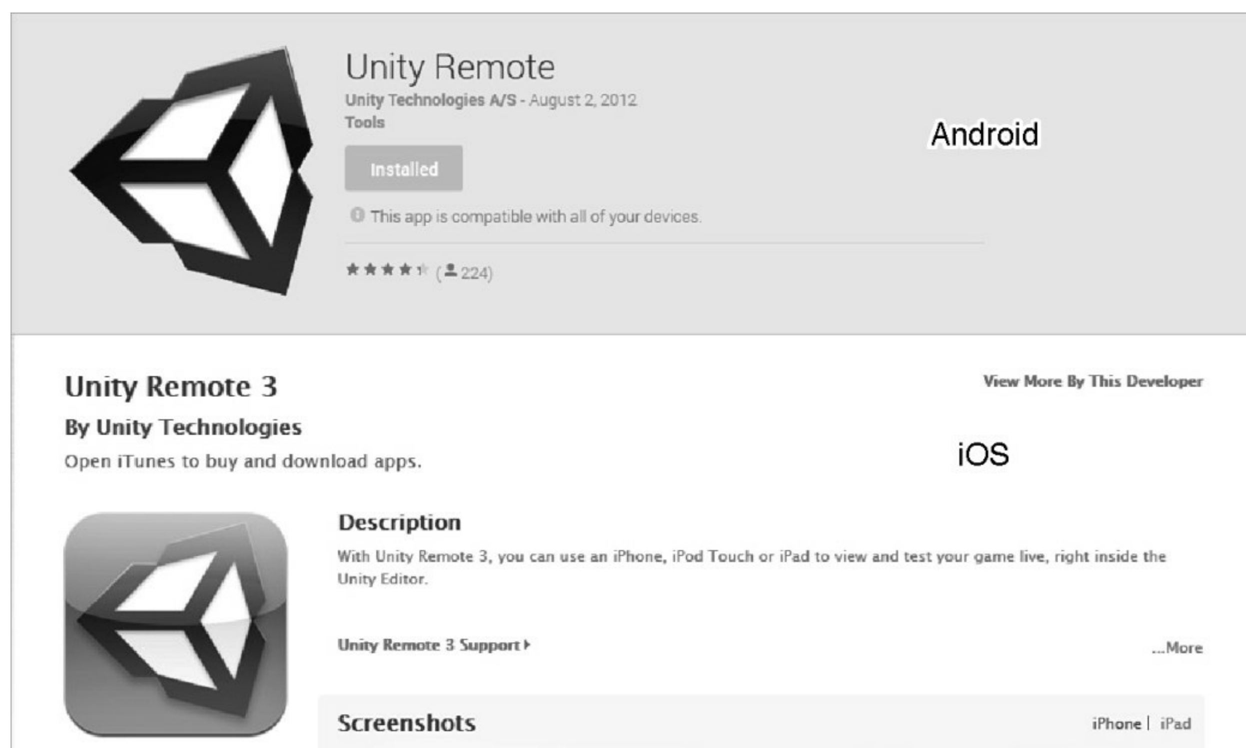


图21.3 不同的应用程序商店

一旦安装，Unity Remote 将同时充当游戏的显示器和控制器。你将能够把单击信息、加速计信息以及多触摸输入发回给Unity。

测试设备的设置

让我们花点时间确保移动开发环境设置正确。在这个练习中，你将从设备使用 Unity Remote，与Unity中的场景交互。如果还没有设置设备，将不能执行所有这些步骤，但是仍然可以通过顺着阅读这些内容理

解所发生的事情。如果这些步骤不工作，它就意味着你的环境没有设置正确。

（1）创建一个新的项目或场景。然后创建一个名为Scripts的新文件夹，并在该文件夹中创建一个名为TestScript的新脚本。

（2）把测试脚本附加到Main Camera 上，并向其中添加以下代码：

```
void OnGUI()
{
  if(GUI.Button(new Rect(Screen.width / 2 - 50, Screen.height / 2 - 50,
100, 100), "Click"))
  {
    camera.backgroundColor =
      new Color(Random.Range(0f,1f),Random.Range(0f,1f),
  Random.
      Range(0f,1f));
  }
}
```

（3）运行场景，并且确保单击按钮将会改变屏幕的背景色。然后停止运行场景。

（4）把移动设备附加到计算机上。一旦计算机识别了设备，就会打开Unity Remote。

（5）再次运行场景。一秒钟后，应该会看到场景的蓝色屏幕和按钮出现在移动设备上。你现在应该能够点按设备屏幕上的按钮，更改场景的背景色。如果发现单击按钮什么也没有发生，可以单击除Game视图之外的其他任何视图。这是由于一个微小的错误引起的，单击Unity中的视图或者某个工具可以修正它。

21.2 加速计

大多数现代的移动设备都带有内置的加速计。加速计会转发关于设备的物理方位的信息，它可以辨别设备是在移动、倾斜还是平放着，也可以在全部3根轴中检测这些方面。图21.4　显示了移动设备的加速计轴以及它们是如何定向的，这称为纵向定向（ portrait orientation）。

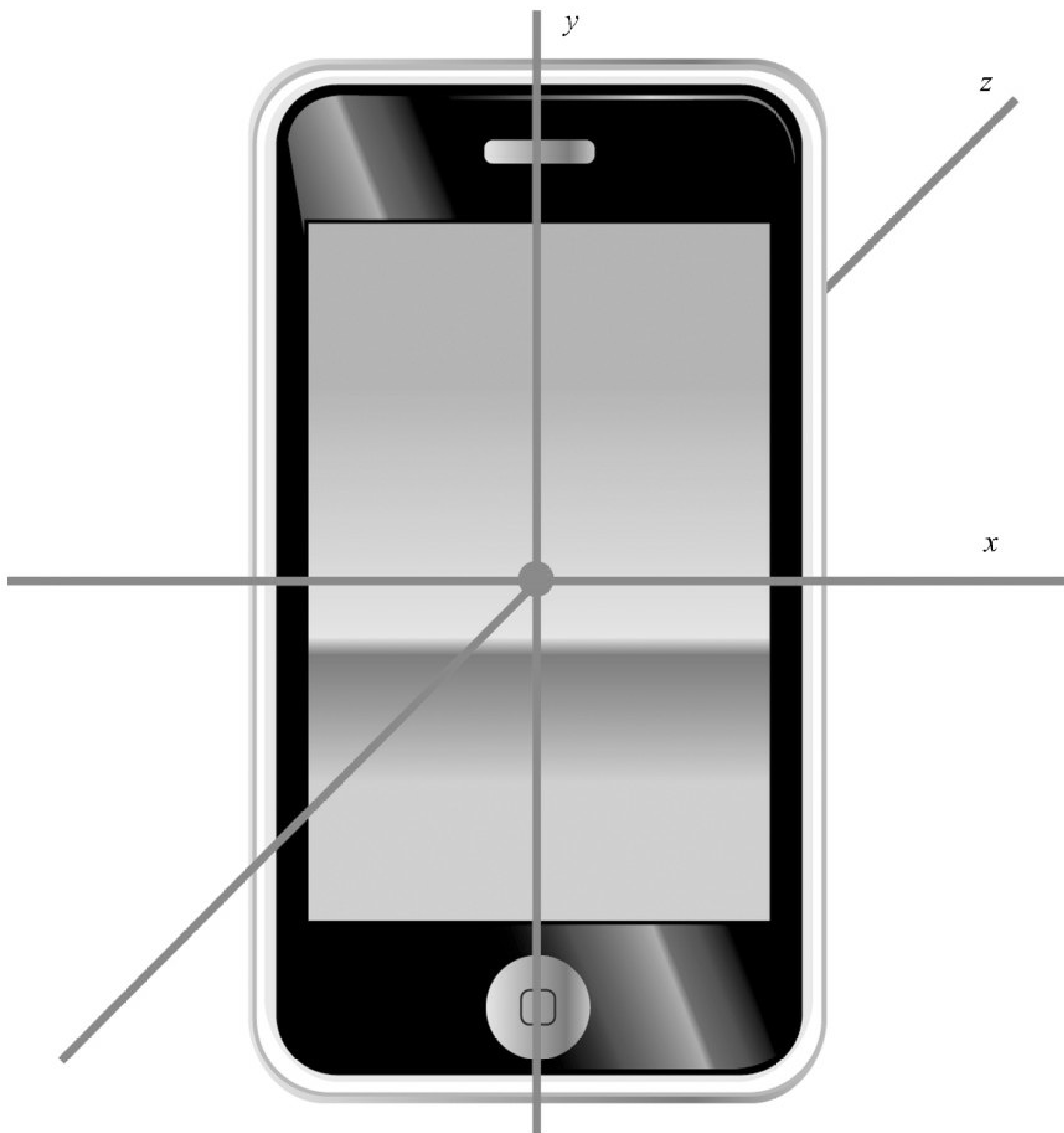


图21.4 加速计轴

在图 21.4 中可以看到，当在正前方垂直握持设备时，设备的默认轴将与 Unity 中的3D 轴对齐。如果转动设备以便在一个不同的方向上使用它，将需要把加速计数据转换成正确的轴。

21.2.1 为加速计设计游戏

在设计游戏以使用移动设备的加速计时，需要记住几件事。第一件事是在任何给定的时间只能可靠地使用加速计的其中两根轴，其原因是无论设备处于什么方位，有一根轴总会主动被重力所吸引。考虑图21.4中的设备的方位，可以看到虽然通过倾斜设备可以操纵x轴和z轴，但是y轴目前正在读取负值（重力正在把它往下拉）。如果将要转动手机，使之面朝上地平放在一个表面上，将只能使用x轴和y轴。在这种情况下，z轴将被主动吸引。

在为加速计设计游戏时，要考虑的另一件事是：输入不是非常准确。移动设备不会以固定的时间间隔从它们的加速计读取数据，通常必须采用近似值。结果就是从加速计读入的输入可能起伏较大，并且不均匀。值得指出的是：不精确程度非常低，尽管如此，它还是存在，因此应该指出来。

21.2.2 使用加速计

就像任何其他的用户输入形式一样，读取加速计输入是通过脚本完成的。你只需从名为acceleration的Vector3变量读取它，该变量是Input对象的一部分。因此，可以通过编写如下代码，访问x轴、y轴和z轴数据：

```
Input.acceleration.x;
Input.acceleration.y;
Input.acceleration.z;
```

使用这些值，可以相应地操纵游戏对象。

注意：

轴不匹配

在结合使用加速计信息与Unity Remote 时，你可能注意到轴与21.2节中描述它们的方式不一致。这是由于 Unity Remote 使游戏的方位基于

所选的屏幕高宽比，这意味着 Unity Remote 将自动以横向方向显示（从一侧握着设备，使得较长的边缘与地面平行），并为你平移轴。因此，当使用Unity Remote时，x轴将沿着设备的长边缘延伸，y轴则沿着短边缘延伸。它似乎有些怪异，但是无论如何，你将有可能像这样使用设备。这可以节省一个步骤。

利用你的意念或手机的动力移动立方体

在这个练习中，将使用移动设备的加速计在场景中四处移动立方体。显然，要完成这个练习，将需要一个带有加速计的移动设备，该设备已配置并且连接好了。

（1）创建一个新的项目或场景，向场景中添加一个立方体，并把它定位于(0, 0, 0)处。

（2）创建一个名为 AccelerometerScript 的新脚本，并把它附加到立方体上。然后把以下代码放在脚本的Update()方法中：

```
float x = Input.acceleration.x * Time.deltaTime;
float z = -Input.acceleration.z * Time.deltaTime;
transform.Translate(x, 0f, z);
```

（3）确保把移动设备插接到计算机上。以纵向方向握持设备，并运行Unity Remote。然后运行场景，注意怎样通过倾斜手机来移动立方体，并且注意手机的哪些轴沿着x轴和y轴移动立方体。

21.2.3 多触摸输入

移动设备倾向于主要受支持触摸的屏幕控制。这些屏幕可以检测何时以及在哪里触摸了它们，它们通常一次可以跟踪多个触摸。触摸的准确次数因设备而异。

触摸屏幕不会只给设备提供一个简单的触摸位置。事实上，关于每个单独的触摸都会存储相当多的信息。在Unity中，每个屏幕触摸都存

储在一个Touch变量中。这意味着每次触摸屏幕时，都会生成一个Touch变量。只要手指保持在屏幕上，Touch变量都将会存在。如果沿着屏幕拖动手指，Touch变量就会跟踪它。这些Touch变量一起存储在一个名为touches的集合中，它是 Input 对象的一部分。如果当前没有什么在触摸屏幕，那么这个触摸的集合将为空。要访问这个集合，可以输入以下代码：

```
Input.touches;
```

使用该集合，可以遍历每个Touch变量，以处理它的数据。执行该操作的代码如下所示：

```
foreach(Touch touch in Input.touches)
{
   //Do something
}
```

如前所述，每个触摸都包含比触摸所在位置的简单屏幕数据更多的信息。表 21.1 包含Touch变量类型的所有属性。

表21.1 触摸属性

属性	描述
deltaPosition	从上一次更新起在触摸位置发生的改变，它可用于检测手指拖动
deltaTime	从上一次更改触摸起经过的时间
fingerId	触摸的唯一索引。例如，它们在设备上的范围是 0～4，一次允许 5 个触摸
phase	触摸的当前阶段。这些阶段可以是 Began、Moved、Stationary、Ended 和 Canceled
position	触摸在屏幕上的 2D 位置
tapCount	在屏幕上执行触摸的点按次数。在编写本书时，只在 iOS 设备上实现了它。在 Android 设备上必须手动管理点按次数

其中每个属性都可用于管理用户与游戏对象之间的复杂交互。

跟踪触摸

在这个练习中，将跟踪手指触摸，并把它们的数据输出到屏幕上。显然，要完成这个练习，需要一个支持多触摸的移动设备，并且已经配

置和连接了它。

（1）创建一个新的项目或场景。

（2）创建一个名为TouchScript　　　的新脚本，并把它附加到Main Camera 上。然后把以下代码放入脚本中：

```
void OnGUI()
{
  foreach(Touch touch in Input.touches)
  {
    string message = "";
    message += "ID: " + touch.fingerId + "\n";
    message += "Phase: " + touch.phase.ToString() + "\n";
    message += "TapCount: " + touch.tapCount + "\n";
    message += "Pos X: " + touch.position.x + "\n";
    message += "Pos Y: " + touch.position.y + "\n";
    int num = touch.fingerId;
    GUI.Label(new Rect(0 + 130 * num, 0, 120, 100), message);
  }
}
```

（3）确保将移动设备插接到计算机上。然后运行场景，利用手指触摸屏幕，并且注意出现的信息，如图21.5所示。移动手指，看看数据是怎样改变的。现在利用多根手指同时触摸，随意地移动它们以及使它们离开屏幕。看看它是怎样独立地跟踪每个触摸的。在屏幕上一次可以获得多少个触摸？

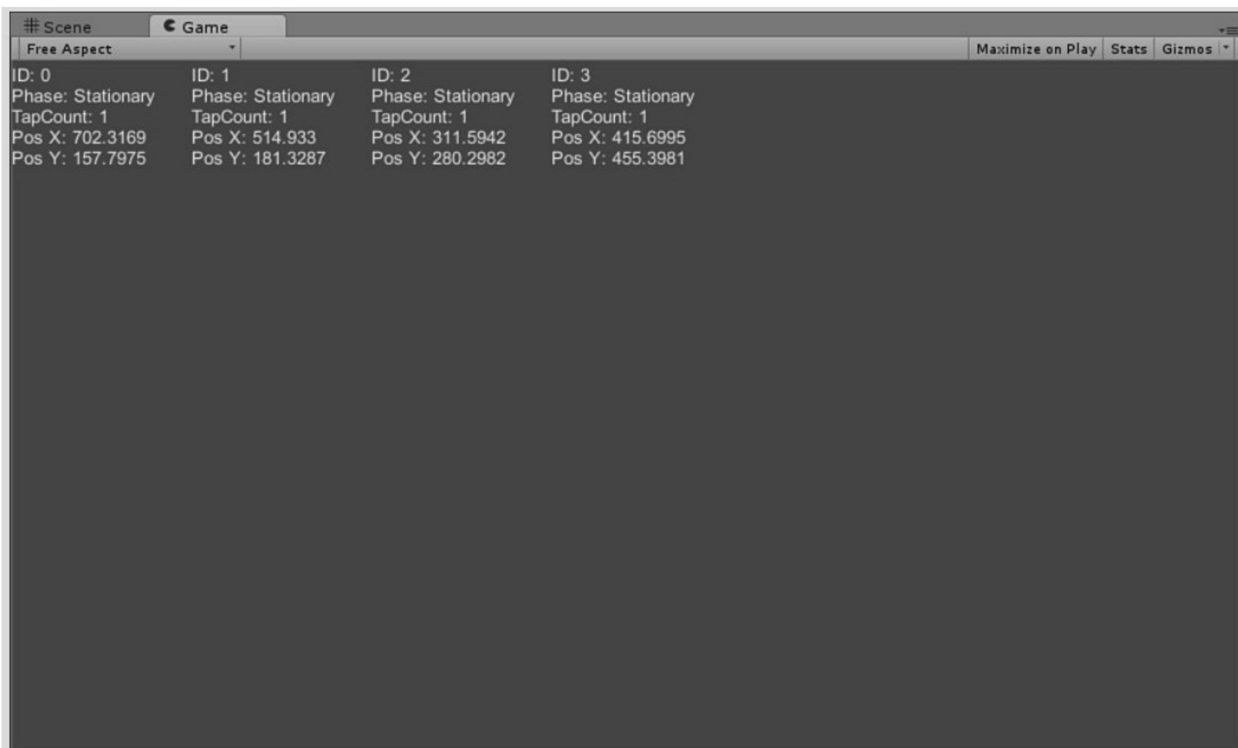


图21.5 屏幕上的触摸输出

警告：

因为我说了而去做，而不是因为我做了再去做！

在上一个练习中，你创建了一个 OnGUI()方法，用于收集关于屏幕上的多个触摸的信息。利用触摸数据构建字符串 message 的代码部分是绝对不可接受的，永远也不要在 OnGUI()方法中执行处理，因为它可能在项目中极大地降低效率。这只是构建示例的最容易的方式，没有不必要的复杂性，并且只用于演示目的。总是要把更新代码保存在属于它的位置，即在Update()方法中。

21.3 小结

在本章中，你学习了在考虑到移动设备的情况下使用 Unity 开发游戏，首先学习了如何配置开发环境以处理Android和iOS。接着，你亲自动手处理了设备的加速计。在本章最后，你试验了Unity的触摸跟踪系统。

21.4 问与答

问：我真的能够只构建游戏一次，就能把它部署到所有主要的平台（包括移动平台）上吗？答：绝对可以！要考虑的唯一一件事是：移动设备一般没有像台式机那样大的处理能力。因此，如果游戏具有许多重型处理或效果，就可能会经历一些性能问题。如果你计划把游戏也部署到移动平台上，就需要确保它高效地运行。

问：iOS设备与Android设备之间的区别是什么？

答：从 Unity 的角度讲，这两种操作系统之间并没有太大的差别，它们都被视作是移动设备。不过，要知道的是，有一些硬件差别可能会影响你的游戏。

21.5 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

21.5.1 问题

1．什么工具允许在Unity运行场景时给它发送实时的设备输入数据？

2．一次可以实际地使用加速计上的多少根轴？

3．一个设备同时可以具有多少个触摸？

21.5.2 答案

1．Unity Remote。

2．两根轴。依赖于你握持设备的方式，第三根轴总会被重力所吸引。

3．它完全依赖于设备。如果设备不支持多触摸，那么它一次就只能具有单个触摸。如果它支持多触摸，就可以具有许多触摸。

21.5.3 练习

在这个练习中，你将基于来自移动设备的触摸输入，在场景中四处移动对象。显然，要完成这个练习，将需要一个支持多触摸的设备，并且配置和连接了它。如果没有该设备，仍然可以顺着阅读下去，以获取基本的思想。在用于第21章（Hour 21）的本书配套资源中可以找到完成的练习，其名称为Hour21_Exercise。

1．创建一个新的项目或场景，并向场景中添加一个定向灯光。

2．向场景中添加3个立方体，把它们分别命名为Cube1、Cube2和Cube3，并且分别定位于(−3, 1, −5)、(0, 1, −5)和(3, 1, −5)处。

3．创建一个名为Scripts的新文件夹，在该文件夹中创建一个名为InputScript的新脚本，并把它附加到3个立方体上。

4．把以下代码添加到脚本的Update( )方法中：

```
foreach(Touch touch in Input.touches)
{
   if(touch.fingerId == 0 && gameObject.name == "Cube1")
      transform.Translate(touch.deltaPosition.x * .05F, touch.
      deltaPosition.y * .05F, 0F);
   if(touch.fingerId == 1 && gameObject.name == "Cube2")
      transform.Translate(touch.deltaPosition.x * .05F, touch.
      deltaPosition.y * .05F, 0F);
   if(touch.fingerId == 2 && gameObject.name == "Cube3")
      transform.Translate(touch.deltaPosition.x * .05F, touch.
      deltaPosition.y * .05F, 0F);
}
```

5．运行场景，并且利用最多3根手指触摸屏幕。注意可以怎样独立地移动3个立方体，还要注意抬起一根手指不会导致其他手指丢失它们的立方体或位置。

第22章 游戏修改

在本章中你将学到：

怎样使Amazing Racer 游戏支持移动；

怎样使Chaos Ball 游戏支持移动；

怎样使Captain Blaster 游戏支持移动；

怎样使Gauntlet Runner 游戏支持移动。

让我们制作许多游戏！更确切地讲，让我们修改你以前制作的游戏，并使它们支持移动开发。你首先将向Amazing Racer游戏中添加移动、查看和跳跃能力。此后，将向Chaos Ball游戏中添加转弯和移动控制。接着，将改变方位，使Captain Blaster游戏以Portrait（纵向）模式工作。最后，将给Gauntlet Runner 游戏添加输入控制。

注意：

完成的游戏

在用于第22章（Hour 22）的本书配套资源中提供了所有完成的移动友好的游戏，它们的名称取自原始的游戏项目。

22.1 Amazing Racer游戏

你制作的第一款游戏可能最难转换到移动设备上。其原因是有3种独特的输入形式：移动、查看和跳跃。移动部分可以轻松地从加速计读取到，但查看和跳跃都来自于触摸输入。因此，需要某种方式来区分触摸输入。

22.1.1 移动和查看

你要做的第一件事是：改变玩家的移动方式。这款特定的游戏利用了第一人称控制器。实现移动控制的最佳方式是打开控制器，并在其内部修改代码。不过，此时，它比想要的更复杂一点，因此你将代之以在已经存在的功能之上创建新的脚本。你将使加速计信息可以向前或向后移动玩家，以及从一边移到另一边。发生的屏幕右半部分的任何触摸都将允许四处移动视图。由于查看（looking）机制的复杂性，向上看和向下看将会旋转控制器的摄像机，而向左看和向右看实际上将旋转控制器自身。

要设置移动和水平查看，可以遵循下面这些步骤。

（1）打开完成的Amazing Racer游戏，向Scripts 文件夹中添加一个名为MobileInputScript的新脚本，并把它附加到Player游戏对象上。

（2）把以下代码添加到脚本中：

```
public float speed = 15;
public float jump = 3;
CharacterController control;
void Start () {
```

```
        control = GetComponent<CharacterController>();
    }
    // Update is called once per frame
    void Update () {
        float x = Input.acceleration.x * Time.deltaTime * speed;
        float z = -Input.acceleration.z * Time.deltaTime * speed;
        transform.Translate(x, 0f, z);
        foreach(Touch touch in Input.touches)
        {
            //turning
            if(touch.position.x > Screen.width / 2)
            {
                transform.Rotate(0f, touch.deltaPosition.x, 0f);
            }
        }
    }
```

（3）把移动设备连接到计算机上，并且运行Unity　Remote。当以Landscape（横向）模式握持设备时，运行场景。注意怎样通过倾斜手机来四处移动。尝试在屏幕的右边拖动手指，注意在屏幕的右边怎样使用触摸来四处查看。

既然已经添加了移动和水平查看，你将希望添加垂直查看。你将把垂直查看组件直接添加到第一人称控制器的摄像机上，它将使用与水平查看相同的机制进行查看。要添加垂直查看，可以遵循下面这些步骤。

（1）在scripts文件夹中添加一个名为MobileLookScript的新脚本，把该脚本附加到Main　Camera上，它嵌套在Player游戏对象下面，如图22.1所示。

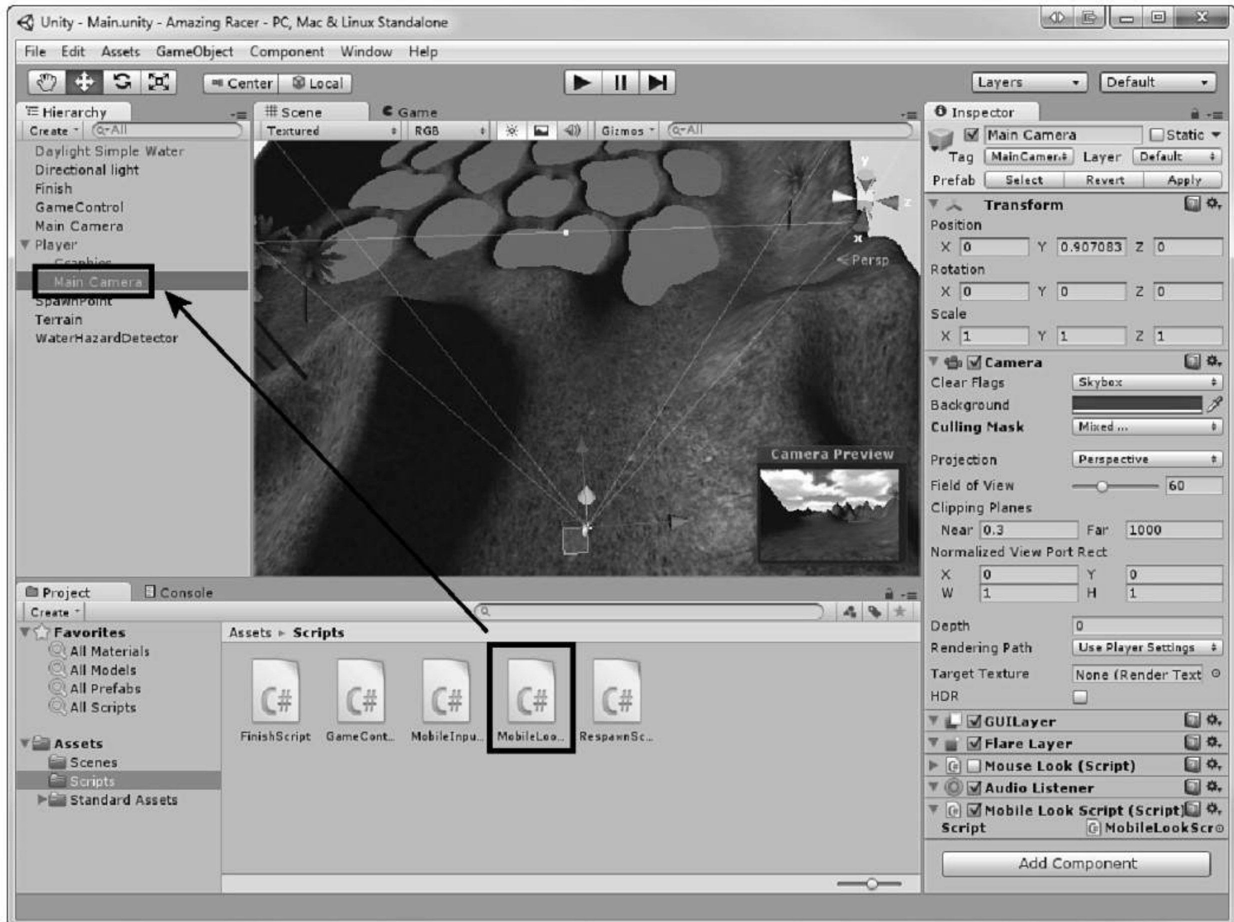


图22.1 把查看脚本添加到控制器的摄像机上

（2）把以下代码添加到查看脚本中的Update()方法中：

```
foreach(Touch touch in Input.touches)
{
    if(touch.position.x > Screen.width / 2)
    {
        transform.Rotate(-touch.deltaPosition.y, 0f, 0f);
    }
}
```

（3）运行场景。注意现在可以在屏幕右边使用手指向上和向下查看。结合使用MobileInputScript与这个脚本，将使你能够使用单独一根手指到处查看屏幕。

22.1.2 跳跃

你想添加到这款游戏中的最后一点功能是跳跃特性。如前所述，这款游戏是利用第一人称控制器制作的，因此，你将添加的跳跃将不会以完全相同的方式工作。执行它的最佳方式是修改控制器代码，以允许跳跃输入。由于这样做比较复杂，你将代之以添加新代码。

跳跃将通过单击屏幕左边的任意位置来工作。在这种情况下，代码将寻找任何点按动作，然后它将使玩家向上跳到空中。为此，可以遵循下面这些步骤。

（1）你将不会创建一个新的脚本。作为替代，将打开以前创建的MobileInputScript脚本。

（2）把以下代码添加到 Update()方法中。你应该已经列出了其中一些代码，它们只是充当一个参考，以使你知道这段代码应该出现在什么位置：

```
foreach(Touch touch in Input.touches)//Here for reference
{
  //jumping
  if(touch.position.x < Screen.width - Screen.width / 2 &&
    touch.phase == TouchPhase.Began)
  {
    control.Move(new Vector3(0f, jump, 0f));
  }
  //turning–Here for reference
  if(touch.position.x > Screen.width / 2)
//...
```

（3）运行场景。注意现在可以怎样利用移动设备跳跃，以及进行移动和查看。

此时，应该为Amazing Racer游戏完成了所有的移动转换工作。

注意：

游戏质量

当在移动设备上运行这款游戏时，你可能注意到它缺少控制质量。跳跃机制有些起伏，移动和转弯机制也有点不稳定。这是正常的，并且无法利用移动设备指示问题。事实是：你正在向一款已经在正常运行的游戏上面添加新功能。并非所有的机制都可以轻松地集成起来，可以对这些机制进行调和，但是这样做将花费太多的时间。记住：你还有3款游戏要处理。作为替代，使用这些游戏指示移动控制可能做什么，并且作为如何实现它们的基本指南。

22.2 Chaos Ball游戏

你创建的第二款游戏Chaos Ball 修改起来更简单一点，其思想同样是用加速计控制球拍的运动，然后可以使用触摸来控制外观。由于不需要其他的输入，将不需要拆分屏幕或者为其他任何方面使用触摸。这里的脚本看上去类似于前一款游戏，因为它们使用类似的控制。要转换这款游戏，可以遵循下面这些步骤。

（1）在Unity中打开Chaos Ball 项目，然后禁用第一人称控制器游戏对象的Main Camera上的Mouse Look (Script)组件，如图22.2 所示。



图22.2 删除 Mouse Look组件

（2）在Scripts文件夹中添加一个名为MouseInputScript的新脚本，

并把该脚本附加到第一人称控制器游戏对象上。然后把以下代码添加到脚本中：

```
public float speed = 15;
void Update (){
    float x = Input.acceleration.x * Time.deltaTime * speed;
    float z = -Input.acceleration.z * Time.deltaTime * speed;
    transform.Translate(x, 0f, z);
    foreach(Touch touch in Input.touches)
    {
        //turning
        if(touch.position.x > Screen.width / 2)
        {
            transform.Rotate(0f, touch.deltaPosition.x * 10, 0f);
        }
    }
}
```

（3）添加另一个名为 MouseLookScript 的脚本，并把它附加到第一人称控制器游戏对象的Main Camera 上，如图22.1 所示。然后把以下代码添加到该脚本的Update()方法中：

```
foreach(Touch touch in Input.touches)
{
    transform.Rotate(-touch.deltaPosition.y, 0f, 0f);
}
```

（4）运行游戏。注意你可以怎样四处移动，以及观察加速计和触摸输入。

这就是把这款游戏转换成移动输入（并使之变得难度更大）所需做的全部工作。要知道的一件事情是：现在玩家可以穿过墙壁以及坠落到

水平面下，这是由于现在是使用Translate()而不是控制器来移动玩家。可以用两种方式修正它：使用已经存在的角色控制器，或者从玩家进行光线投射，以便查看玩家是否离墙太近。

22.3 Captain Blaster游戏

Captain Blaster 游戏在你迄今为止制作的游戏当中比较独特，这是由于它被视作是一款2D游戏，并且是一款垂直定向的游戏。这意味着你不必关心z轴，它还意味着需要配置游戏，使之以Portrait（纵向）模式工作。回忆可知，当竖着握持设备时，如果使它的短边缘与地面平行，同时使它的长边缘与地面垂直，就会获得纵向方向（portrait orientation）。

要使Unity以Portrait（纵向）模式工作，将需要对Unity编辑器执行一些修改。你必须告诉它你希望以Portrait（纵向）模式工作。为此，可以遵循下面这些步骤。

（1）打开Unity项目Captain Blaster。

（2）选择File > Build Settings 命令，调出Build Settings 对话框。不要太关心这里有什么，下一章中将更广泛地介绍它。在左边的菜单中，依赖于你使用的设备类型，可以选择Android或iOS，然后单击Switch Platform。完成后，可以单击右上角的“X”，退出这个对话框，如图22.3所示。如果你是在Mac机器上操作，将在左上角具有类似的按钮。

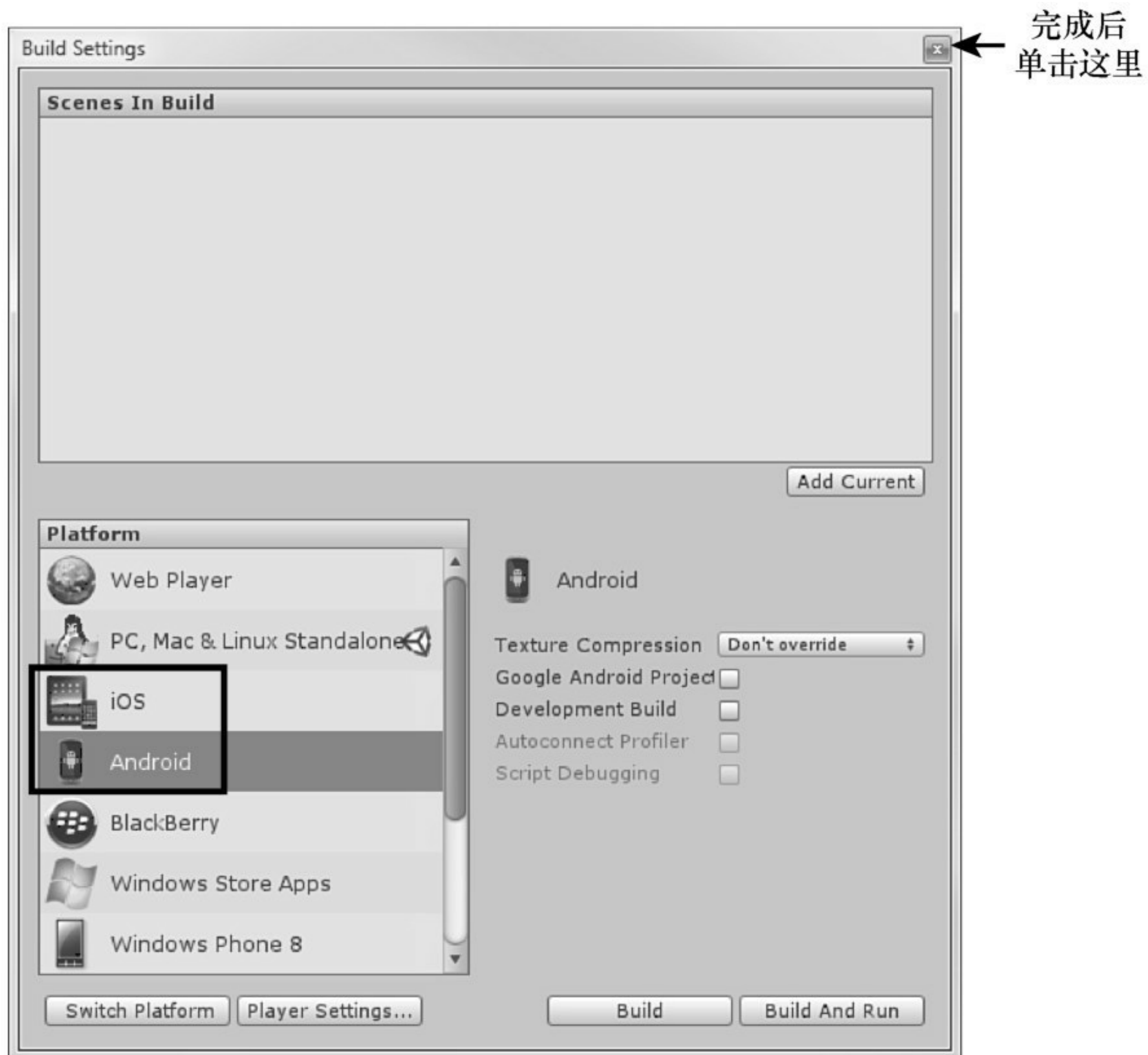


图22.3 切换平台

（3）在Game视图中，选择游戏的分辨率。如果选择你使用的移动设备的尺寸，那就是最好的。如果不知道这些尺寸是什么，可以选择通用的3:2 Portrait，如图22.4 所示。

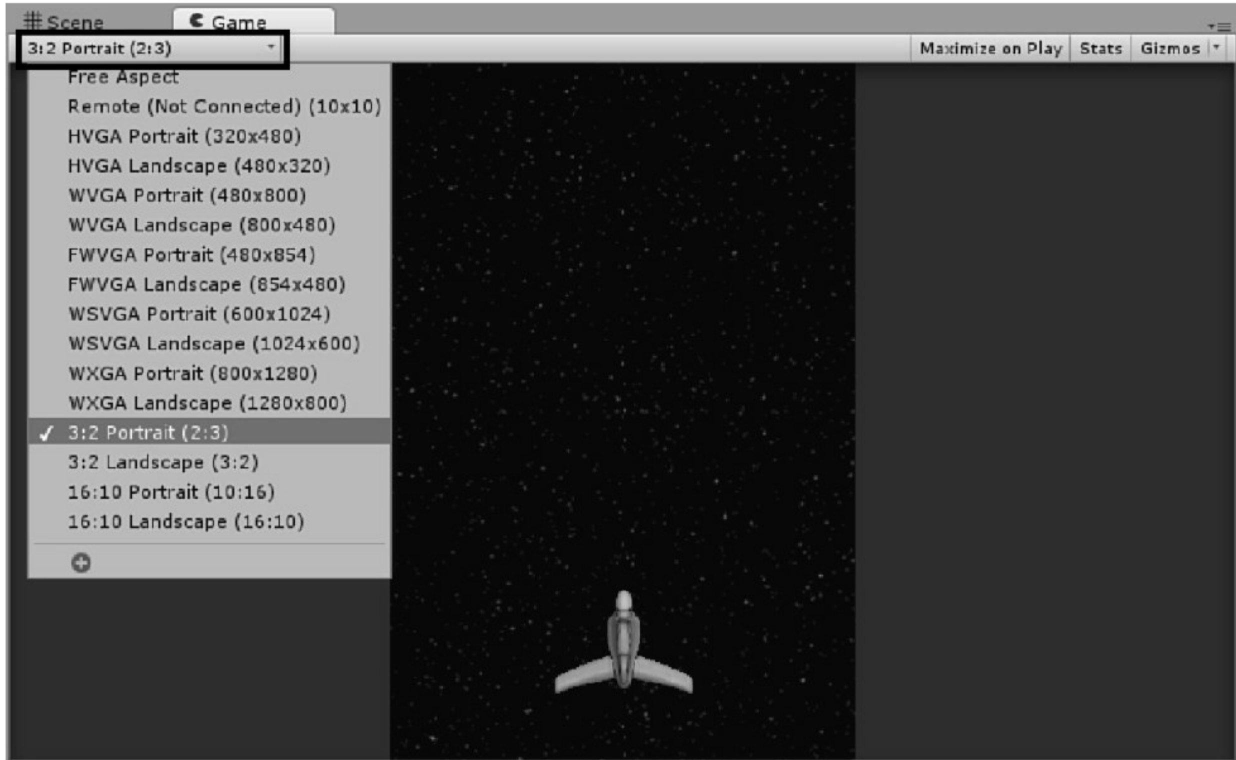


图22.4 更改场景的分辨率

注意：

控制台错误

在这个过程中，你可能在控制台中看到一堆的红色错误。此刻，可以安全地忽略这些错误。它们是在转换场景中的纹理和模型时生成的，一旦播放场景，它们就会消失，并且一切都会进行自我校正。

现在，就准备好播放场景。当在 Unity Remote 中运行时，应该以适当地纵向显示它。现在，只需把加速计和点按映射到游戏输入。这一次它将很容易，因为在你最初制作这款游戏时，并没有使用内置的控制器之一。现在可以只修改你编写的代码，以利用移动设备。

（1）在Scripts文件夹中，找到PlayerScript并打开它。

（2）为了使移动工作，可以把下面一行代码：

```
transform.Translate(Input.acceleration.x * speed * Time.deltaTime, 0f, 0f);
```

放在这一行代码之下：

```
transform.Translate(Input.GetAxis("Horizontal")* speed * Time.deltaTime, 0f, 0f);
```

（3）为了使射击工作，可以把下面一行代码：

```
if(Input.GetButtonDown("Jump"))
```

修改为：

```
if(Input.GetButtonDown("Jump")|| Input.touches.Length > 0)
```

现在就设置了移动输入。你可能注意到留下了原始控制，这样很好，因为它允许游戏同时在移动设备和计算机上运行。要提到的一件事是：由于把分辨率修改得更窄，许多流星将在屏幕外再生。为了改进这一点，将需要修改流星可以再生的区域的尺寸。

22.4 Gauntlet Runner游戏

你制作的最后一款游戏Gauntlet Runner也最容易转换到移动设备上。为了使该游戏工作，只需把应用于Captain Blaster 游戏的代码修改也应用于这款游戏即可。由于这比较烦人，你将实现一种不同的系统，使玩家能够四处移动。

这一次，玩家将通过左右拖动他们的手指，来实现左右移动。为了进行跳跃，玩家必须向上轻拂他们的手指。这两个动作都依赖于Touch变量内的deltaPosition变量。为了实现它，可以遵循下面这些步骤。

（1）在Unity中打开Gauntlet Runner项目。

（2）在Scripts文件夹内找到并打开PlayerScript脚本。修改Update()方法，使之包含以下代码：

```
transform.Translate(Input.GetAxis("Horizontal") * Time.deltaTime *
strafeSpeed, 0f, 0f);
//if there is a touch
if(Input.touches.Length > 0)
{
  //use the position of the first one
  transform.Translate(Input.touches[0].deltaPosition.x *
Time.deltaTime *strafeSpeed,
    0f, 0f);
}
if(transform.position.x > 3)
  transform.position = new Vector3(3, transform.position.y, transform.
  position.z);
```

```
else if(transform.position.x < -3)
    transform.position = new Vector3(-3, transform.position.y, transform.
position.z);
if (anim.GetCurrentAnimatorStateInfo(0).IsName("Base Layer.Jump"))
{
    anim.SetBool("Jumping", false);
    jumping = true;
}
else
{
    jumping = false;
    if(Input.GetButtonDown("Jump"))
    {
        anim.SetBool("Jumping", true);
    }
    //check for "flick" if there are touches
    else if(Input.touches.Length > 0)
    {
    if(Input.touches[0].deltaPosition.y > 2)
        anim.SetBool("Jumping", true);
    }
}
```

（3）运行游戏。注意来回滑动手指将使玩家从一边移动到另一边，还要注意怎样通过向上轻拂手指从而能够进行跳跃。

在处理用于这款游戏的代码时，你可能会注意到下面一行代码：

```
if(Input.touches[0].deltaPosition.y > 2)
```

你可能感到迷惑的是，为什么在这里使用值 2。其基本思想是：尽

管你可能非常快地向上轻拂手指，游戏每秒钟还是循环60次。这意味着与游戏相比，你的动作实际上相当慢。因此，用于确定你是否正在轻拂手指的值非常低。如果把这个值设置得较高，游戏可能不会识别较慢的轻拂动作；如果把它设置得较低，在你没有轻拂手指时，游戏反而可能认为你在这样做。

22.5 小结

在本章中，你重新构建了以前的4款游戏，以包括进移动设备控制。从 Amazing Racer游戏开发，你向其中添加了移动、跳跃和查看控制。接着，你修改了Chaos Ball 游戏，允许移动加速计和触摸输入。你修改了下一款游戏Captain Blaster 中的装备，其中把游戏修改成移动的纵向方向，还添加了加速计移动和点按射击。在最后一款游戏Gauntlet Runner 中，你试验了一种新式的控制，并添加了轻扫和轻拂动作，用于控制玩家。

22.6 问与答

问：一些游戏似乎不能很好地转换为移动的，这正常吗？

答：是的，很正常。通常，如果在制作游戏时没有考虑到移动平台，将难以转换它。与简单的移动设备相比，计算机和游戏控制台具有更多可供它们使用的控制选项。在设计游戏时，如果移动版本在将来有可能实现，那么总要问问自己是否要考虑到它。

22.7 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

22.7.1 问题

1．在Amazing Racer游戏中怎样处理把屏幕同时用于跳跃和查看的需要？

2．在Chaos Ball 游戏中使用平移来移动玩家的主要问题是什么？

3．判断题：Captain Blaster游戏被修改成横向方向。

4．在Gauntlet Runner游戏中什么构成了“轻拂”动作？

22.7.2 答案

1．一半屏幕用于跳跃输入，另一半则用于查看。

2．玩家能够穿过墙壁，以及坠落到游戏世界之下。

3．错误。它被修改成纵向方向。

4．当玩家确实非常快地向上移动他们的手指时，就产生了轻拂动作。

22.7.3 练习

在本章中，你在4款不同的游戏上处理了许多控制机制。在开发游戏时，有时可能会对主要机制进行彻底革新。更改输入或控制的工作方式以便尝试构建更好用户体验的情况并不鲜见。与任何重大的修改一样，过后总是要重新测试，以便查看所做的修改具有什么影响。对于这个练习，要回过头去重新玩这些游戏。这一次，要利用移动控制玩游

戏。像以往一样，要记录下你喜欢什么以及你不喜欢什么。玩完游戏后，要把这些记录与你最初制作游戏时所做的记录做比较（你保存了那些记录，对吗？）。看看你可以对控制或游戏执行哪些修改，以便获得更好的移动体验。尝试实现其中一些或所有的修改。

第23章 润色和部署

在本章中你将学到：

怎样管理游戏中的场景；

怎样保存场景之间的数据和对象；

不同的玩家设置；

怎样部署游戏。

在本章中，你将学习对游戏进行润色并部署它，首先将学习如何在不同的场景之间游走。然后，你将探索在场景之间保存数据和游戏对象的方式。接着，你查看了 Unity 玩家及其设置，然后学习了如何编译和部署游戏。

23.1 管理场景

迄今为止，你在 Unity 中所做的一切事情都发生在同一个场景中。尽管以这种方式构建大型、复杂的游戏肯定是可能的，但是使用多个场景一般要容易得多。场景背后的思想是：它是游戏对象的自含式集合。因此，当在场景之间转换时，所有现有的游戏对象都会被销毁，并且会创建所有新的游戏对象。不过，可以阻止这种情况发生，下一节中将讨论它。

注意：

再论场景是什么？

在本书前面已经讨论了场景是什么。不过，现在应该利用你目前拥有的知识再次讨论这个概念。理想情况下，场景就像是游戏中的一个关卡。不过，对于难度总是越来越大或者动态生成关卡的游戏，并不一定是这样。因此，比较好的做法可能是把场景视作一个公共资源列表。由使用相同对象的许多关卡组成的游戏实际上可能只包含一个场景。仅当需要清除一串对象并加载一串新对象时，新场景的思想才确实变得是必要的。实质上讲，不要仅仅由于你能够这样做，就把关卡拆分到不同的场景中。仅当玩游戏和资源管理需要时，才创建新的场景。

23.1.1 建立场景顺序

在场景之间转换相对比较容易，它只需要一点点设置即可正常工作。你要做的第一件事是把项目的合适场景添加到项目的编译设置中，如下所示。

（1）单击File > Build Settings 命令，打开编译设置。

（2）打开Build Settings 对话框，单击你希望出现在最终项目中的任何场景，并把它们拖到Scenes In Build 窗口中，如图23.1所示。

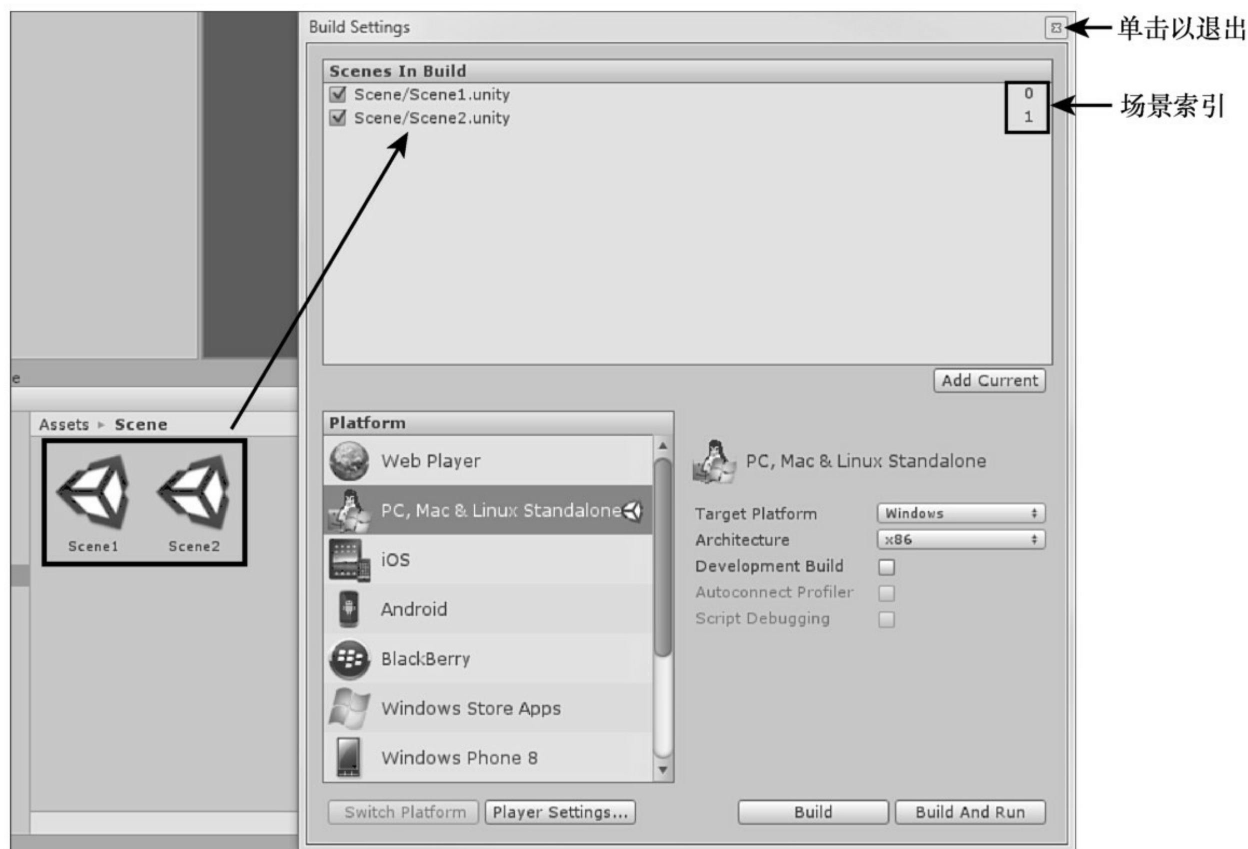


图23.1 把场景添加到编译设置中

（3）注意出现在Scenes In Build 窗口中的场景旁边的数字，以后将用到它们。完成后，单击右上角的“X”，退出窗口。如果你使用的是Mac，那么在左上角将具有类似的按钮。

现在就可以引用以及更改场景。

把场景添加到编译设置中

在这个练习中，将把场景添加到项目的编译设置中。要保存你在这里创建的项目，下一节将会使用它。

（1）创建一个新项目，并在Assets下面添加一个名为Scenes的新文件夹。

（2）单击File > New Scene命令，创建一个新场景，然后单击File >

Save Scene 命令保存它。在Scenes文件夹中把该场景另存为Scene1。重复这个步骤，保存一个Scene2场景。

（3）打开编译设置（单击File > Build Settings 命令），先把Scene1拖到Scenes In Build窗口中，然后把Scene2拖入其中。Scene1应该具有索引0，Scene2应该具有索引1。保存这个项目，以便以后使用。

23.1.2 切换场景

既然已经建立了场景顺序，在它们之间进行切换就很容易。要更改场景，可以使用LoadLevel( )方法，它是Application 对象的一部分。该方法接受单个参数，它要么是一个代表场景索引的整数，要么是一个代表场景名称的字符串。因此，要加载一个名称为GameOverScene并且索引为4的场景，可以编写下面两行代码之一：

```
Application.LoadLevel（4）; //Load by index
Application.LoadLevel(GameOverScene); //Load by name
```

这个方法调用将立即销毁所有现有的游戏对象，并加载下一个场景。注意：这个命令会立即执行并且不可逆，因此在调用它之前要确保它就是你想要做的。

通过按钮更改场景

在这个练习中，将使用一个GUI（graphical user interface，图形用户界面）按钮在两个场景之间进行切换。这个练习需要用到在本章前面创建的项目，如果还没有完成它，现在就要这样做，然后才能继续下面的操作。一定要保存这个项目；在下一节中将再次使用它。

（1）加载以前创建的项目。在Scenes文件夹中找到Scene1文件，双击以加载它。然后向场景中添加一个立方体，并把它置于(0, 0, 0)处。

（2）创建一个名为Scripts的文件夹，并向其中添加一个名为LoadSceneTwo的脚本。然后把该脚本附加到Main Camera 上，并把以下

代码添加到其中：

```
void OnGUI()
{
   if(GUI.Button(new Rect(5, 5, 100, 100), "Load Scene2"))
   {
      Application.LoadLevel（1）;
   }
}
```

（3）保存场景（单击File > Save Scene 命令），然后打开Scene2（如果你不记得怎么做，可以查看第（1）步）。在Scripts文件夹中创建一个名为LoadSceneOne的新脚本，然后把它附加到Main Camera上，并把以下代码添加到其中：

```
void OnGUI()
{
   if(GUI.Button(new Rect(5, 5, 100, 100), "Load Scene1"))
   {
      Application.LoadLevel(0);
   }
}
```

（4）保存场景，并再次加载Scene1。运行场景，注意现在可以通过单击出现在每个场景上的按钮，在两个场景之间进行转换。还要注意立方体只存在于 Scene1 中，在 Scene2中销毁了它。

23.2 保留数据和对象

既然你已经学习了如何在场景之间进行切换，那么无疑就会注意到在切换期间没有转移数据。事实上，迄今为止所有的场景都完全是自含式的，无需保存任何内容。不过，在更复杂的游戏中，保存数据（通常称为持久性，persisting）就变得确实非常必要。在本节中，你将学习如何保存从一个场景到另一个场景的对象，以及如何把数据保存到文件以便以后使用。

23.2.1 保存对象

在场景之间保存数据的一种容易的方式是：保存带有活动数据的对象。例如，如果具有一个玩家对象，它上面具有一些脚本，其中包含生命、存货清单、分数等信息，那么确保将这么大的数据量转移到下一个场景中的最容易的方式是：确保它们不会被销毁。有一种容易的方式可以实现这一点，它涉及一个名为 DontDestroyOnLoad( )的方法。该方法接受单个参数，它是你想保存的游戏对象。因此，如果你想保存一个存储在名为 Brick 的变量中的游戏对象，可以编写以下代码：

```
DontDestroyOnLoad(Brick);
```

由于该方法接受一个游戏对象作为参数，对于对象来说，另一种使用它的极佳方式是使用this关键字调用它自身。为了使对象保存它自身，可以把以下代码放入附加到对象上的脚本的Start( )方法中：

```
DontDestroyOnLoad(this);
```

现在，当切换场景时，保存的对象将在那里等待。

保留对象

在这个练习中，你从一个场景到另一个场景保存了一个立方体。这个练习需要用到在本章前面创建的项目。如果你还没有完成它，现在就要这样做，然后才能继续下面的操作。一定要保存这个项目，下一节将再次使用它。

（1）加载以前创建的项目。确保Scene1是当前加载的场景，注意立方体存在于你以前创建的场景中。

（2）在Scripts文件夹中创建一个名为DontDestroyScript的新脚本。把该脚本附加到立方体上，并利用以下代码替换Start()方法：

```
void Start ()
{
  DontDestroyOnLoad(this);
}
```

（3）保存并运行场景。注意：现在当你切换场景时，立方体将保持不变。现在会在场景之间保留立方体。一定要保存这个项目，以便将来使用。

23.2.2 保存数据

有时，需要把数据保存到一个文件中，以便将来访问。你可能需要保存的一些内容有：玩家的分数、配置参数或者存货清单。当然还有许多复杂的、功能丰富的方式可用于保存数据，但是有一种简单的解决方案，它被称为PlayerPrefs。PlayerPrefs是一个对象，用于把基本的数据在本地保存到你的系统上的一个文件中，并在以后可以使用PlayerPrefs取回这些数据。

把数据保存到 PlayerPrefs，就像为数据提供某个名称以及提供数据本身一样简单。你用于保存数据的方法依赖于数据的类型。例如，要保存一个整数，可以调用 SetInt( )方法。要获取这个整数，可以调用

GetInt( )方法。因此，用于把值10 作为分数保存到PlayerPrefs 并取回该值的代码将如下所示：

```
PlayerPrefs.SetInt("score", 10);
PlayerPrefs.GetInt("score");
```

同样，还有用于保存字符串和浮点数的方法，它们分别是SetString( )和SetFloat()。使用这些方法，可以轻松地把想要的任何数据保存到一个文件中。

使用PlayerPrefs

在这个练习中，将把数据保存到PlayerPrefs文件中。这个练习需要用到在本章前面创建的项目。如果你还没有完成它，现在就要这样做，然后才能继续下面的操作。

（1）打开你在前面创建的项目，并且加载了Scene1。向Scripts文件夹中添加一个名为SaveData 的新脚本，并把它附加到Main Camera 上，然后把以下代码添加到该脚本中：

```
string playerName = "";
void OnGUI()
{
    playerName = GUI.TextField(new Rect(5, 120, 100, 30), playerName);
    if(GUI.Button(new Rect(5, 180, 50, 50), "Save"))
    {
        PlayerPrefs.SetString("name", playerName);
    }
}
```

（2）保存 Scene1 并加载 Scene2。创建一个名为 LoadData 的新脚本，并把它附加到Main Camera 上，然后把以下代码添加到脚本中：

```
string playerName = "";
```

```
void Start()
{
    playerName = PlayerPrefs.GetString("name");
}
void OnGUI()
{
    GUI.Label(new Rect(5, 120, 50, 30), playerName);
}
```

（3）保存 Scene2 并重新加载 Scene1。运行场景，在文本框中输入你的名字，并单击Save 按钮。现在单击Load Scene2 按钮，加载Scene2。注意你输入的名字是怎样写到屏幕上的。将数据保存到PlayerPrefs中，然后在不同的场景中从PlayerPrefs重新加载它。

警告：

数据安全

尽管使用 PlayerPrefs 保存游戏数据非常容易，但它并不是非常安全。数据存储在玩家的硬盘驱动器上的一个未加密的文件中。因此，玩家可以轻松地打开文件，并操纵其中的数据。这可能给他们提供一种不公平的优势，或者会破坏游戏。要知道的是，顾名思义，PlayerPrefs 打算用于保存玩家的参数设置。它就是这样发生的，以至于可用于其他的方面。真正的数据安全实现起来很困难，肯定超出了本书的范围。只要知道：PlayerPrefs目前适合于你的需要，但是在将来，你将深入研究更复杂的、安全的保存玩家数据的方式。

23.3 Unity玩家设置

Unity 提供了多种设置，一旦编译了游戏，它们就会影响游戏的工作方式。这些设置称为玩家设置（player settings），它们管理像游戏的图标和支持的屏幕高宽比这样的事情。有许多种设置，其中又有许多是自解释的，但是要花时间彻底检查它们，并且了解它们可以做什么。可以单击Edit > Project Settings > Player 命令，打开Player Settings 窗口，该窗口将在Inspector视图中打开。

23.3.1 跨平台的设置

你将看到的第一批设置是跨平台的设置，如图23.2所示。无论为哪个平台（Windows、iOS、Android、Mac 等）构建游戏，这些设置都适用于它。本节中介绍的大多数设置都是自解释的。产品名称是显示为游戏标题的名称。图标应该是任何有效的纹理图像文件。注意：图标的尺寸必须是2的方幂，如8×8、16×16、32×32、64×64等。如果图标与这些尺寸不匹配，缩放可能不会正确地工作，并且图标质量可能非常低。还可以在Cursor设置中指定自定义的光标，并且定义光标热点的位置。

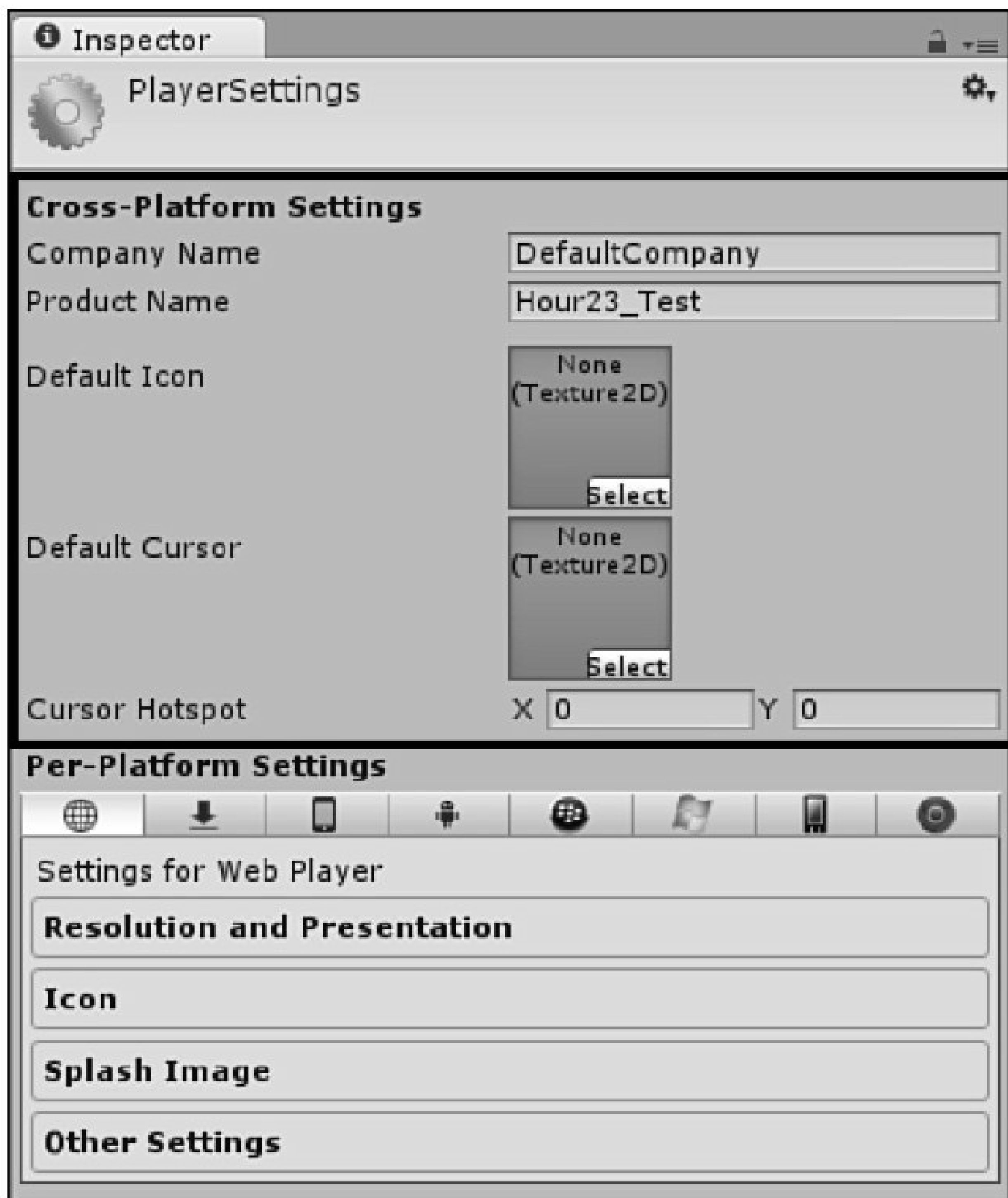


图23.2 跨平台的设置

23.3.2 每个平台的设置

每个平台的设置是指特定于不同平台的不同设置。即使本节中有多个重复的设置，但你仍然必须为不同的平台单独设置它们。可以从选择栏中选择特定平台的图标，来选择该平台，如图23.3所示。

其中许多设置都需要更特定地理解正在构建游戏的平台。直到你更好地理解了那个特定平台的工作方式之后，才应该修改这些设置。其他设置相当直观，仅当需要尝试实现特定的目标时，才需要修改它们。例如，Resolution和Presentation设置处理的是游戏窗口的尺寸。对于桌面编译，它们可能是窗口或者全屏幕风格的，支持广泛不同的屏幕高宽比。通过启用或禁用不同的屏幕高宽比，可以允许或禁止玩家在玩游戏时选择不同的分辨率。

如果在Cross-Platform Settings 区域中为Default Icon 属性指定了一幅图标图像，就会为你自动填充图标设置。可以看到，将基于所提供的单个图像来生成图标图像的不同大小。这就是为什么使提供的图像具有正确的尺寸很重要的原因。还可以在开机画面（splash image）设置中为游戏提供一个开机画面，开机画面是在实际的玩家第一次启动游戏时添加到 Player Settings对话框中的图像。

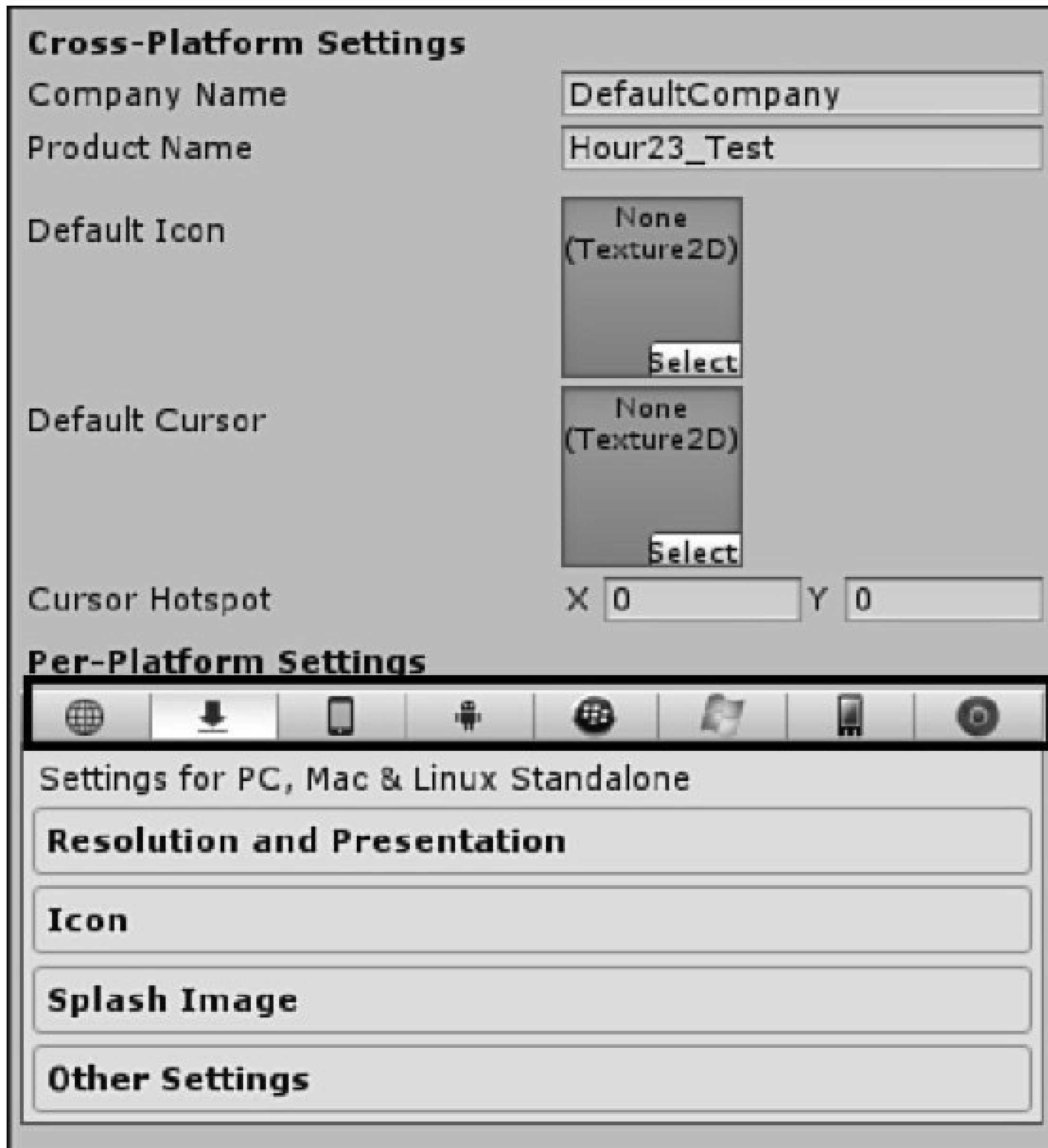


图23.3 平台选择栏

注意：

太多的设置

你可能注意到Player Settings中有大量的设置没有在本节中介绍。事实是大多数属性已经设置为默认值，使得你可以快速编译游戏。其他设置全都用于实现高级功能或润色。对于大多数设置，如果你不理解它们用于做什么，就不应该处理它们，因为它们可能导致怪异的行为，或者

根本就会阻止游戏工作。简而言之，在你更熟悉游戏编译概念以及你使用的不同特性之前，目前可以只使用更基本的设置。

注意：

太多的玩家

本章中大量使用了玩家（player）这个术语，因为可以用两种方式使用这个术语。显然，第一种是实际玩游戏的玩家，这是一个人。可以使用这个术语的第二种方式是描述Unity Player。Unity Player 是玩游戏的窗口（就像电影播放器或电视一样）。这存在于计算机（或设备）上。因此，当你听到“玩家”这个词时，它可能意指一个人；但是当你听到 Player Settings时，它可能意指实际地显示游戏的软件。

23.4 编译游戏

假定您已经构建好了你的第一款游戏，完成了所有的工作并在编辑器中对各个方面都进行了测试。你彻底检查了Player Settings，并且按你想要的方式设置了所有的选项。现在就应该编译游戏。在这个过程中，你需要知道两个设置窗口。第一个是Build Settings 窗口，你将在其中确定编译过程的最终结果；第二个是Game Settings 窗口，这些设置会被实际的玩家看到，它们指示了玩家如何选择分辨率和控制配置。

23.4.1 Build Settings窗口

Build Settings 窗口包含编译游戏的条件，在这里指定编译游戏的平台以及游戏中的各个场景。你以前见过一次这个对话框，但是现在应该更详细地讨论它。

要打开BuildSettings对话框，可以单击File>BuildSettings命令。一旦BuildSettings对话框打开，就可以像你所想的那样更改和配置游戏。图23.4显示了Build Settings对话框以及它上面的各个项目。

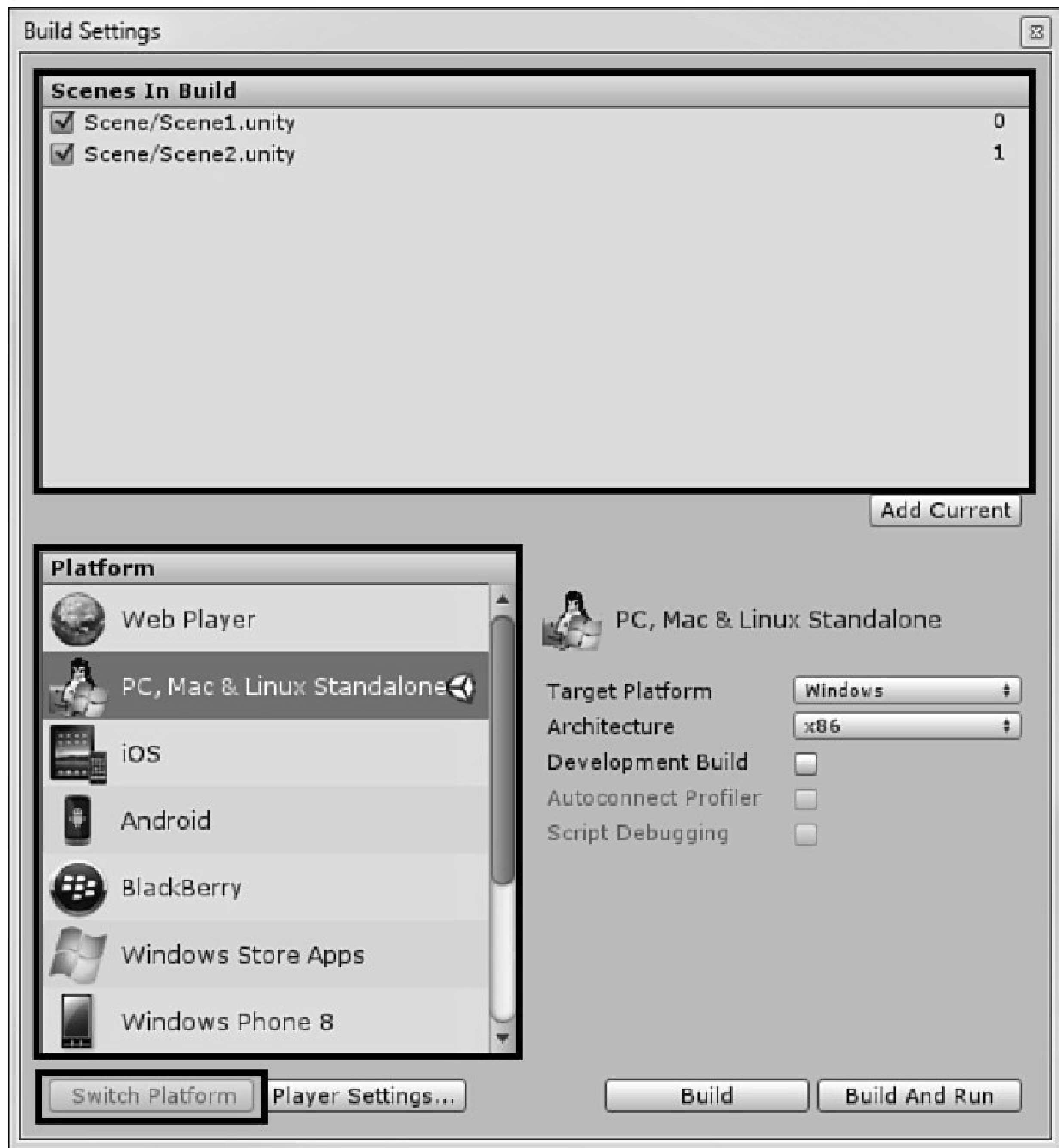


图23.4 Build Settings 对话框

可以看到，在Platform区域中，可以指定为之编译游戏的新平台。如果选择一个新平台，将需要单击Switch Platform来进行切换。单击Player Settings按钮，在Inspector 视图中打开Player Settings 对话框。你以前见过Scenes In Build区域，在这里确定哪些场景将转变成游戏以及

它们的顺序。对于所选的特定平台，还具有多个不同的编译设置。
PC、MAC & Linux Standalone 设置比较简单，应该是自解释的。唯一要注意的是Development Build 选项，它将允许游戏在运行时带有调试器和性能监测器（专业特性）。

当准备好编译游戏时，可以单击Build 只编译游戏，或者单击Build and Run 在编译完后运行游戏。Unity创建的文件将依赖于所选的平台。

23.4.2 Game Settings窗口

当从其实际的文件（而不是从 Unity 内）运行编译过的游戏时，将给玩家展示一个Game Settings 对话框，如图 23.5 所示。从这个对话框中，玩家可以为他们的游戏体验选择选项。



图23.5 Game Settings 对话框

你最先可能注意到的是游戏的名称，它出现在窗口的标题栏中。此外，你在Player Settings对话框中提供的任何开机画面都将出现在这个窗口的顶部。在第一个选项卡Graphics中，玩家可以指定他们想要用于玩游戏的分辨率。可用的分辨率列表是由你在 Player Settings 对话框中允许或禁止的屏幕高宽比确定的。玩家也可以选择在窗口或全屏幕中运行游戏，并且可以选择他们的质量设置。

玩家然后可以转换到Input选项卡，如图23.6所示。在这个选项卡上，玩家可以把任何输入轴重新映射到他们想要的按钮上。

注意：

告诉你就是这样！

你可能想起在本书前面通知你：总是应该设法确保从玩家读取的输入基于输入轴之一，而不是特定的键。这是为什么呢？如果寻找特定的键而不是轴，那么玩家除了使用你计划的控制模式之外将别无选择。如果你认为这不算太糟糕，只需记住有许多人（如残疾人）使用非标准的输入设备。如果你拒绝了他们重新映射控制的能力，那么他们也许不能玩你的游戏。使用轴代替特定的键对你来说需要做的工作微不足道，但是可以产生玩家喜欢或憎恨你的游戏的区别。

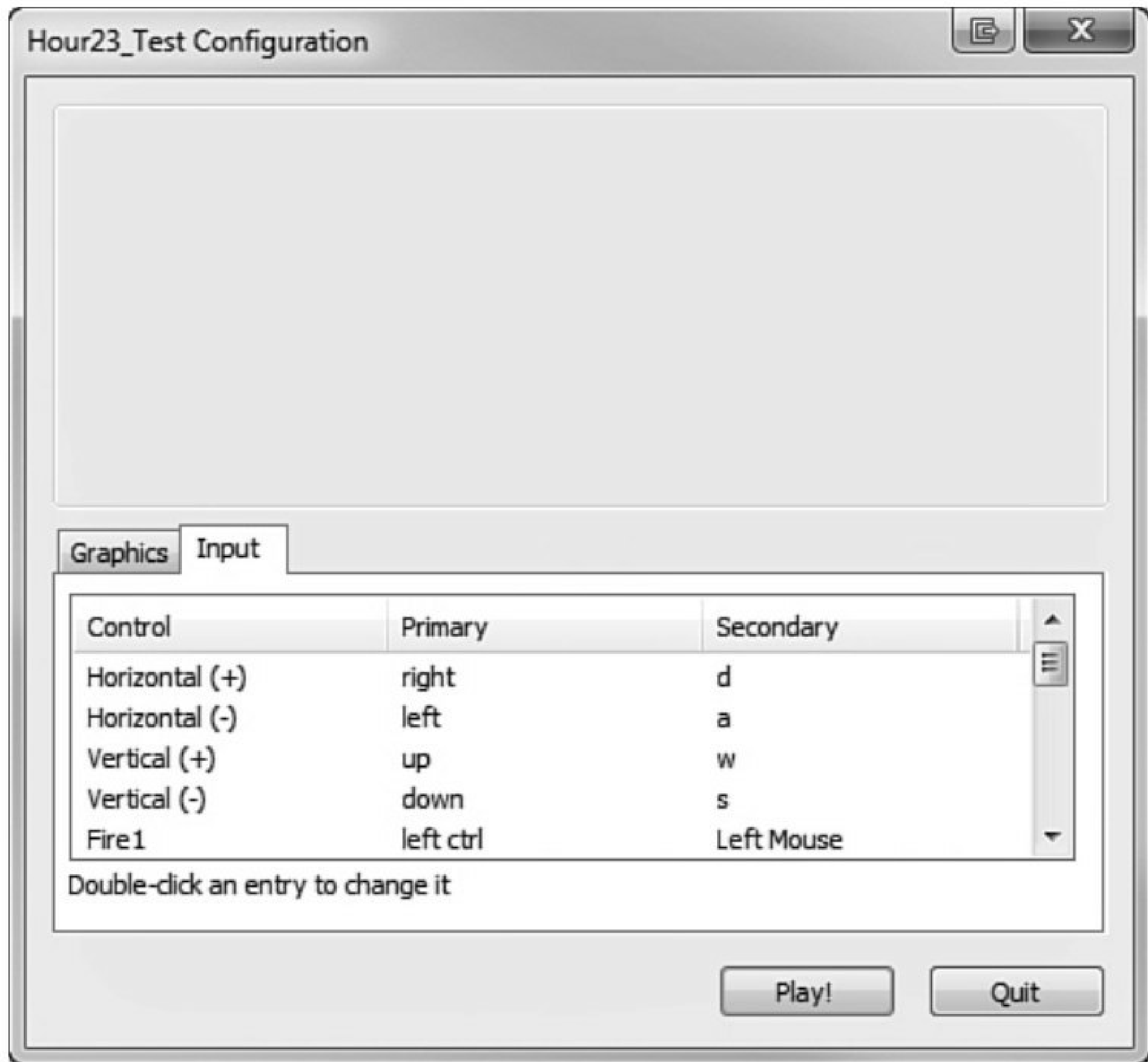


图23.6 输入设置

在玩家选择了他们想要的设置之后，他们就可以单击 Play!按钮，终于可以开始享受你的游戏了。

23.5 小结

在本章中，你学习了在 Unity 中润色和编译游戏。你首先学习了如何使用 LoadLevel( )方法在 Unity 中更改场景。接着，你学习了如何保留游戏对象和数据。然后，你学习了多种玩家设置。最后，你学习了编译游戏来结束本章。

23.6 问与答

问：有许多设置看上去很重要，为什么没有介绍它们？

答：说实话，大多数设置对你来说都是不必要的。事实是：它们都不重要……直到它们变得重要为止。大多数设置都是特定于平台的，并且超出了本书的范围。与其花大量篇幅仔细讨论你可能从来都不使用的设置，不如在你需要时留给你自己去学习它们。

23.7 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

23.7.1 问题

1．怎样确定游戏中的每个场景的索引？

2．判断题：可以使用PlayerPrefs对象保存数据。

3．游戏的图标应该具有什么尺寸？

4．判断题：游戏设置中的输入设置允许玩家重新映射游戏中的所有输入。

23.7.2 答案

1．在把场景添加到Scenes In Build 的列表中之后，它们将具有分配给它们的索引。

2．正确。

3．游戏图标应该是一个正方形，它的各条边的长度是2的幂，如8×8、16×16、32×32等。

4．错误。玩家只能重新映射基于输入轴（而不是特定的按键）建立的输入。

23.7.3 练习

在这个练习中，你将为桌面操作系统编译一款游戏，并且试验多个不同的特性。这个练习本身并没有太多的事情要做，你应该把大部分时间用于试验不同的设置，并且观察它们的效果。由于这只是一个让你编

译游戏的示例，因此在本书配套资源中没有提供完成的项目让你参考。

1．选择你以前创建的任何项目，或者创建一个新项目。

2．进入Player Settings 对话框，并且按你想要的任何方式配置玩家。

3．进入Build Settings 对话框，并且确保把场景添加到Scenes In Build 列表中。确保选择PC, MAC & Linux Stan dalone 平台，然后编译游戏。

4．找到你编译的游戏文件并运行它。试验不同的游戏设置，看看它们会怎样影响人们玩游戏。

第24章 结束语

在本章中你将学到：

你迄今为止完成了什么工作；

从这里去往何处；

你可以使用什么资源。

在本章中，你将结束 Unity 学习之旅。你首先将准确查看自己迄今为止所做的事情。接着，你将了解自己可以去往何处，以继续改进自己的技能。然后，将向你介绍多个可用的资源，它们可以帮助你继续学习。

24.1 成果

当你花了大量的时间做某件事情时，有时可能会忘记一路上所完成的一切工作。反省你在开始学习时所拥有的技能并把它们与你现在所拥有的技能做比较将是有帮助的。在你的发现之旅中可以找到许多动力与满足感。下面让我们先探讨一些数字。

24.1.1 19小时的学习时间

首先并且最重要的是，你花了19 小时（可能更多）的时间认真学习了利用Unity 4进行游戏开发的多个元素。下面列出了你学过了一些内容。

如何使用Unity编辑器以及它的许多窗口和对话框。

关于游戏对象、变换和变形。你学习了2D与3D 坐标系统，以及局部坐标系统和世界坐标系统。你变成了使用Unity内置的几何形状的专家。

关于模型。确切地讲，你学习了模型怎样由应用于材质的纹理和着色器组成，材质反过来又会应用于网格。你学习了网格由三角形组成，它们包含3D空间里的许多点。

怎样在 Unity 中构建地形。你雕刻了独特的地形，并学习使用了一些工具，它们是构建任何类型的游戏所需要的。你利用周围的效果和环境细节改进了那些游戏世界。

关于摄像机和灯光的所有知识。

在 Unity 中编程。如果你在阅读本书之前从未编写过程序，这就是一件重要的事。祝你好运！

关于碰撞、物理材质和光线投射。换句话说，你将在通过物理学进行对象交互方面迈出第一步。

关于预设和实例化。

如何使用Unity内置的GUI 控件构建GUI。你已经学习过GUI 代表什么。

如何通过Unity的角色控制器控制玩家。在它的基础上，你构建了一个自定义的2D角色控制器用于自己的项目。

如何使用多种粒子系统制作出色的粒子效果。你学习了使用新的Shuriken 系统，但是还试验了一些遗留系统的效果。

遗留的动画系统。这包括学习动画的详细解释，并且对它们的制作方式有了一点了解。

如何使用Unity新增的Mecanim动画系统。在学习它时，你学习了怎样重新映射模型上的绑定，以使用不是专门为它制作的动画。你还学习了如何编辑动画，从而制作你自己的动画剪辑。

怎样在项目中操纵音频。你学习了如何处理2D和3D音频，以及如何循环播放和交换音频剪辑。

怎样处理为移动设备制作的游戏。你学习了怎样利用Unity Remote测试游戏，以及利用设备加速计和多触摸屏幕。

怎样使用多个场景和数据持久性对游戏进行润色。你学习了如何编译并且玩游戏。

这个列表相当长，它甚至还不完整。在你从头至尾阅读这个列表时，我希望你记得体验和学习其中的每一项。你已经学到了许多知识！

24.1.2 4款完整的游戏

在学习本书的过程中，你创建了4款游戏：Amazing Racer、Chaos Ball、Captain Blaster和Gauntlet Runner。你设计了所有这些游戏，仔细

研究了理念，确定了规则，并且提出了需求。一旦完成了这些，你就构建了游戏的所有实体。接着，你明确地把每个对象、玩家、游戏世界、球、流星等放入游戏中。你编写了所有的脚本，并把所有的交互性构建到游戏中。然后，最重要的是，你测试了所有的游戏，并确定了它们的长处和弱点。你玩过这些游戏，并且让自己的伙伴也一起玩这些游戏。你考虑了如何改进它们，并且甚至尝试自己改进它们。下面探讨一下你使用的一些机制和游戏理念。

Amazing Racer 游戏：一款相对于时钟的3D竞走游戏。这款游戏利用了内置的第一人称角色控制器以及完全雕刻的、纹理化的地形，并且使用了水障、触发器和灯光。

Chaos Ball 游戏：另一款与Chaos 同名的3D游戏。这款游戏具有大量的碰撞和物理动力学。你利用了物理材质构建一个有弹性的舞台，甚至实现了角上的球门，它们可以把特定的对象转变成运动学。

Captain Blaster游戏：一款复古风格的2D太空射击游戏。这是第一款使用滚动背景和 2D 效果的游戏，也是第一款你制作的玩家可以输掉的游戏。第三方模型和纹理确保这款游戏具有高级的图形样式。

Gauntlet Runner 游戏：一款具有 3D 特征的赛跑游戏，其中你不得不收集充电装置以及避开障碍物。这款游戏利用了Mecanim动画和第三方模型，并且聪明地操纵了纹理坐标以实现3D滚动效果。

不要忘记你还回过头来修改了每一款游戏，以使它们能够在移动设备上工作。你获得了设计游戏、编译游戏、测试游戏以及为新硬件更新游戏的经验。

24.1.3 58个场景

在学习本书的过程中，你遵照书中的指导创建了 58 个场景。让我们稍微思考一下这个数字。这意味着在通读本书时，你已经亲自实践了

至少58个不同的理念，这可以让你获得相当多的经验。

目前，你可能领会了本节的要点。你做了许多事情，并且应该为之感到自豪。你亲自使用了Unity游戏引擎的很大一部分，在前进道路上，这些知识可以给你很大的帮助。

24.2 从这里去往何处

即使你学完了本书，也与完成有关制作游戏的学习相距甚远。事实上，可以相当准确地说，在像这样一个快速发展的行业中，永远都学无止境。即便如此，这里还是给出了一些建议，指出你接下来可以做什么。

24.2.1 制作游戏

不，严肃地讲，是制作游戏。不能把它夸大其事。如果你正尝试学习关于 Unity 游戏引擎的更多知识，尝试找到一份与游戏有关的工作，或者有一份这样的工作但是指望获得更好的工作，就请制作游戏吧。在游戏（或者任何软件）行业的新手当中，一种常见的错误观念是：只有知识可以使你获得一份工作或者改进你的技能。我们不能离真相越来越远，经验是最重要的因素，请动手制作游戏吧。它们甚至不必是大型游戏。从制作几款较小的游戏（比如你在本书中制作的游戏）开始。事实上，尝试立即制作大型游戏可能会导致挫败和沮丧。不过，无论你决定做什么，都要制作游戏。

24.2.2 与人打交道

有许多本地和在线协作组希望为企业或出于娱乐目的而制作游戏。事实上，如果它们拥有像你一样具有丰富的 Unity 经验的人才，那么它们将很幸运。记住，你已经自己开发了 4款游戏。与其他人合作可以教会你许多有关组动态的知识。此外，与其他人合作还允许你在可以制作的游戏中实现更高级别的复杂性。设法找到美术师和音响师，使你的游

戏充满丰富的媒体效果。你将发现在团队中工作是了解自身长处和弱点的最佳方式。它可以对现实情况进行非常好的检查，并且极大地提升你的自信心。

24.2.3 记录

把你的游戏以及游戏开发历程记录下来，可能会给你的个人发展带来极大的方便。无论你打算开始写博客，还是只想保存一份个人笔记，你的观察资料都可以在当下以及你的回忆中起到很好的作用。记录也可以是一种磨练技能以及与他人合作的极佳方式。通过把你的思想放在那里，你可以收到反馈，并通过其他人输入的信息来学习。

24.3 可供使用的资源

一般来讲，在你继续学习 Unity 游戏引擎以及进行游戏开发的过程中，有许多资源可供你使用。首先并且最重要的是 Unity 文档。这份手册是关于 Unity 的方方面面的官方资源。有一个站点（http://docs.unity3d.com）通过一种技术方法介绍了Unity，知道这一点很重要。不要把这个站点看作是一种学习工具，因为它只是一份手册。

Unity还在它们的Learn站点上提供了五花八门的在线培训，可以从http://unity3d.com访问这个站点。在该站点上，可以找到许多视频、项目及其他资源，它们有助于改进你的技能。

如果你发现自己具有不能利用这两个资源回答的问题，Unity 社区将非常有帮助。可以在http://forums.unity3d.com上找到一些论坛。在这里，你可以参与一般性的交谈和广泛的问题。还有Unity Answers 站点，其地址是http://answers.unity3d.com。在这里可以询问具体的问题，并从Unity专业人员那里获得直接的答案。

除了官方 Unity 资源之外，还有几个游戏开发站点可供你使用，其中两个更流行的站点是http://www.gamasutra.com和http://www.gamedev.net。这两个站点都具有大型社区，并且会定期发表文章。它们的主题并不仅限于Unity，因此它们可以提供非常大的、无倾向性的信息源。

24.4 小结

在本章中，你回顾了迄今为止所做的一切事情，并且还展望了未来。你首先检查了在学习本书的过程中完成的所有事情。然后，探讨了在这里可以做的一些事情，以继续改进你的技能。最后，介绍了Internet上可供你使用的一些免费资源。

24.5 问与答

问：在阅读本章的内容之后，我不能提供帮助，但是可以感觉到你认为我们应该制作游戏，是这样吗？

答：是的。我相信自己提过它几次。我不能足够地强调通过实践和创意继续磨练你的技能有多重要。

24.6 测验

花一些时间做完这里的问题，确保你完全掌握了本章所学的知识。

24.6.1 问题

1．哪一款游戏涉及3D项目收集？

2．你应该对自己迄今为止所完成的事情感到自豪吗？

3．为了继续提升你的游戏开发技能，你可以做的单独一件最好的事情是什么？

4．有多少个Unity社区站点可供你使用？

24.6.2 答案

1．Gauntlet Runner游戏。

2．绝对应该。

3．你可以制作游戏！

4．有两个这样的站点：Unity论坛和Unity Answers。

24.6.3 练习

本书最后一章背后的中心思想是回顾和巩固你所学的知识，本书的最后一个练习也遵循了相同的中心思想。撰写所谓的项目总结（post-mortem）在游戏业中很常见，项目总结背后的思想是：撰写一篇关于你制作的游戏的文章，目的是让其他人阅读它。在项目总结中，你将分析哪些方面工作得很好，哪些方面则不然。你的目的是把自己发现的陷阱告知其他人，使得他们不再会落入相同的陷阱中。

在这个练习中，你将撰写一份关于你制作的游戏之一的项目总结。你不必让任何人阅读它，撰写它是因为它很重要。一定要花一些时间完成这项工作，因为你将来可能会再次阅读它。你将感到惊奇的是，你发现有些事情比较困难，而有些事情则令人愉悦。

在撰写了项目总结之后，把它打印出来（除非它是手写的），并把它放入本书中。以后，当你再次碰到本书时，一定要打开项目总结并阅读它。